VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

SIMULACE DISTRIBUOVANÝCH SYSTÉMŮ
DISTRIBUTED SYSTEMS SIMULATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Anton Ďuriš**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Radomil Matoušek, Ph.D.**
SUPERVISOR

BRNO 2021

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání diplomové práce

Ústav:	Ústav automatizace a informatiky
Student:	**Bc. Anton Ďuriš**
Studijní program:	Strojní inženýrství
Studijní obor:	Aplikovaná informatika a řízení
Vedoucí práce:	**doc. Ing. Radomil Matoušek, Ph.D.**
Akademický rok:	2020/21

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Simulace distribuovaných systémů

Stručná charakteristika problematiky úkolu:

Modelování chování distribuovaných systémů pomocí Petriho sítí s důrazem na studium stochastických jevů. Distibuované systémy nabývají stále většího významu v rámci velkých aplikací (sociální sítě, vědecké výpočty, podnikové systémy, velké e–shopy agregující více prodejců) a výpočetních systémů (cluster, grid, cloud).

Cíle diplomové práce:

1/ Seznámení se stochastickými Petriho sítěmi a způsoby modelování distribuovaných systémů.
2/ Implementace modelů distribuovaných systémů:
– horizontálně škálovaná webová aplikace rozdělená na více služeb s distribuovanou databází (přetížení, selhávání komponent, kaskádovité jevy v systému)
– velký výpočetní systém typu grid, například platforma BOINC s projektem Folding@home (deadline pro výpočty, pauza výpočtu uživatelem počítače)
3/ Analýza výsledků simulací a různých scénářů pro dostupnost a propustnost aplikace a vlivů na dokončení výpočetních úloh.

Seznam doporučené literatury:

BAUSE, F., KRITZINGER, P. S. , 2002. STOCHASTIC Petri Nets. In Stochastic Petri Nets. Vieweg+Teubner Verlag. https://doi.org/10.1007/978-3-322-86501-4.

HORVÁTH, Gábor, Telek MIKLÓS a Bruno SERICOLA, 2014. Analytical and Stochastic Modelling Techniques and Applications: 21st International Conference, ASMTA 2014, Budapest, Hungary, June 30 -- July 2, 2014,Proceedings. Imprint: Springer. Programming and Software Engineering, 8499. ISBN 9783319082196.

AJMONE MARSAN, M., c1995. Modelling with generalized stochastic Petri nets. New York: Wiley. ISBN 0471930598.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2020/21

V Brně, dne

L. S.

doc. Ing. Radomil Matoušek, Ph.D. ředitel ústavu	doc. Ing. Jaroslav Katolický, Ph.D. děkan fakulty

Abstrakt

Táto práca sa zaoberá modelovaním distribuovaných systémov pomocou Petriho sietí. Distribuované systémy sa stále viac implementujú do aplikácií a výpočtových systémov, pričom ich úlohou je zabezpečiť dostatočný výkon a stabilitu pri veľkom množstve jej používateľov. Pri modelovaní distribuovaných systémov je dôležité stochastické správanie Petriho sietí, ktoré zabezpečí reálnejšie simulácie. Preto sa táto práca zameriava hlavne na časované Petriho siete. V teoretickej časti tejto práce sú zhrnuté distribuované systémy, ich vlastnosti, typy a dostupné architektúry, a tiež Petriho siete, ich reprezentácia, typy a princíp fungovania. V praktickej časti boli implementované dva modely, a to horizontálne škálovaná webová aplikácia rozdelená na viac služieb s distribuovanou databázou a veľký sieťový výpočtový systém, presnejšie platforma *BOINC* s projektom *Folding@home*. Oba modely boli implementované pomocou knižnice *PetNetSim* jazyka *Python*. Cieľom tejto práce bolo vykonanie simulácií na vytvorených modeloch pre rôzne scenáre ich správania.

ABSTRACT

This thesis is focused on distributed systems modeling using Petri nets. Distributed systems are increasingly being implemented in applications and computing systems, where their task is to ensure sufficient performance and stability for a large number of its users. When modeling a distributed systems, stochastic behavior of Petri nets is important, which will provide more realistic simulations. Therefore, this thesis focuses mainly on timed Petri nets. The theoretical part of this thesis summarizes distributed systems, their properties, types and available architectures, as well as Petri nets, their representation, types and the principle of an operation. In the practical part, two models were implemented, namely a horizontally scaled web application divided into several services with a distributed database and a large grid computing system, more precisely the *BOINC* platform with the *Folding@home* project. Both models were implemented using the *PetNetSim* library of *Python*. The goal of this thesis is to perform simulations on the created models for different scenarios of their behavior.

Kľúčové slová

distribuované systémy, modelovanie distribuovaných systémov, simulácia distribuovaných systémov, Petriho siete, časované Petriho siete, stochastické procesy, sieťové výpočty, BOINC, PetNetSim, Python

KEYWORDS

distributed systems, distributed systems modeling, distributed systems simulation, Petri nets, timed Petri nets, stochastic processes, grid computing, BOINC, PetNetSim, Python

ÚSTAV AUTOMATIZACE A INFORMATIKY

2021

BIBLIOGRAFICKÁ CITÁCIA

ĎURIŠ, Anton. *Simulace distribuovaných systémů*. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky, 2021, 79 s. Diplomová práca. Vedúci práce: doc. Ing. Radomil Matoušek, Ph.D.

Vyhlásenie autora o pôvodnosti diela

Prohlašuji, že tato práce je mým původním dílem, vypracoval jsem ji samostatně pod vedením vedoucího práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury.

Jako autor uvedené práce dále prohlašuji, že v souvislosti s vytvořením této práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků.

V Brně dne 21. 5. 2021 .....

Anton Ďuriš

Poďakovanie

Ďakujem vedúcemu diplomovej práce pánovi doc. Ing. Radomilovi Matoušekovi, Ph.D. a školiteľovi Ing. Ladislavovi Dobrovskému za poskytnutie odbornej pomoci, skúseností a cenných rád pri tvorbe tejto práce. Ďakujem tiež mojej rodine a priateľom za podporu počas štúdia.

Obsah

1 Úvod

Posledných pár rokov nastal veľký technický pokrok pri internetových sieťach a hardvérovom vybavení rôznych zariadení. Tento pokrok spôsobil, že sa takéto zariadenia s pripojením na internetovú sieť rozšírili po celom svete. Tieto zariadenia môžu byť rôznych veľkostí, od stolových počítačov, až po malé senzorové zariadenia. Pri tomto rozšírení ale nastáva problém s dostupnosťou už vytvorených aplikácií, ktoré neboli stavané na dnešnú záťaž. K vyriešeniu tohto problému prispel vývoj distribuovaných systémov, ktoré dokážu pomocou komunikácie viacerých serverov poskytovať takúto aplikáciu pri veľkom počte jej používateľov. V dnešnej dobe je ďalší problém aj výpočtový výkon, pretože existujú úlohy, ktoré sú náročné na spracovanie. Distribuované systémy poskytujú nástroje, ako získať výkon superpočítača, a to zosieťovaním viacero počítačov, ktoré budú spolupracovať pre dostiahnutie spoločného cieľa.

Distribuované systémy sa teda skladajú z veľkého množstva zariadení a dajú sa charakterizovať touto vetou: "Distribuovaný systém je zbierka autonómnych výpočtových jednotiek, ktoré sa javia ich používateľom ako jeden súdržný systém."[1] Veľká výhoda takýchto systémov sú ich vlastnosti, ktoré zabezpečujú rozširovanie systému bez zmeny na zariadeniach.

Pre modelovanie distribuovaných systémov v tejto práci boli použité Petriho siete, ktoré umožňujú reprezentovať systémy pomocou grafov a následne na nich vykonať simuláciu. Distribuované systémy pozostávajú najmä zo stochastických javov, preto sa táto práca zameriava hlavne na časované Petriho siete. Takéto siete umožňujú simuláciu pre náhodné časy spracovania úloh v distribuovanom systéme, a tak poskytujú vernejšie a reálnejšie výsledky. Existuje viacero nástrojov pre vytvorenie Petriho sietí, ale táto práca sa zameriava na knižnicu *PetNetSim*[2] jazyka *Python*. Táto knižnica bola vybraná hlavne pre integráciu časovaných Petriho sietí a jednoduchosť ich implementácie.

Táto práca sa delí na dve časti, a to teoretickú a praktickú časť. V teoretickej časti tejto práce sú zhrnuté distribuované systémy a Petriho siete. Pri distribuovaných systémoch je uvedená ich definícia, vlastnosti, možné architektúry a typy takýchto systémov. Pri Petriho sieťach je popísaná ich grafická reprezentácia a práca s komponentami siete. Dôraz sa berie na časované Petriho siete, ktoré tvoria hlavný prostriedok pre vytvorenie modelov v praktickej časti. Ďalej sú uvedené triedy objektov knižnice *PetNetSim*, ich metódy a použitie.

V praktickej časti boli vytvorené dva modely distribuovaných systémov. Prvý model je implementácia horizontálne škálovanej webovej aplikácie, ktorá je rozdelená na viac služieb s distribuovanou databázou. Takéto aplikácie sú napríklad sociálne siete, streamovacie služby a podobne. Druhý model obsahuje implementáciu veľkého

sieťového výpočtového systému. Ide o systém na platforme *BOINC* s projektom *Folding@home*. Takýto systém poskytuje pomocou množstva zosieťovaných počítačov veľký výpočtový výkon, ktorý je v prípade tohto projektu poskytnutý na simuláciu skladania proteínov a ich pohyb, ktoré sa vyskytujú pri rôznych chorobách.

Cieľom tejto práce bolo okrem vytvorenia modelov aj ich simulácia a analýza jej výsledkov. Oba namodelované systémy sa môžu dostať do rôznych kritických situácií, ktoré boli simulované pomocou zmeny parametrov Petriho sietí vytvorených modelov. Analýza bola vykonaná pomocou viacerých behov simulácií a ich následného štatistického spracovania.

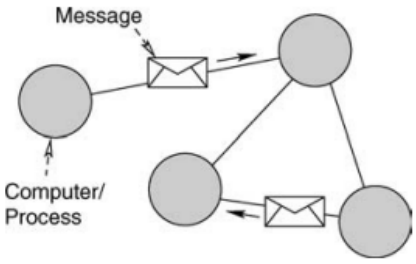
2 Distribuované systémy

V tejto kapitole si bližšie priblížime pojem distribuovaný systém. Prvé počítače zaberali veľa miesta, boli drahé a pracovali samostatne jeden od ďalšieho. Zmena vlastností prvých počítačov prišla s pokrokom v technológii a to hlavne v dvoch oblastiach. Začalo to v osmdesiatych rokoch, kedy nastal vývoj výkonnejších mikroprocesorov, od 8-bitových a neskôr 16-, 32- a 64-bitových, ktoré sú dnes súčasťou už takmer každého počítača a iných zariadení. Takéto mikroprocesory majú väčšinou viac výpočtových jadier, a preto sa prispôsobuje aj vývoj programov pre využitie paralelizmu. Dôsledkom tohoto vývoja a zmenšovaním komponentov vieme dosiahnuť vysoký výpočtový výkon vo veľmi malom zariadení, akým môže byť napríklad smartfón. Ďalšou oblasťou bol vývoj vysokorýchlostnej počítačovej siete. Pomocou lokálnej siete LAN sme schopní prepojiť stovky počítačov v obmedzenej vzdialenosti, napríklad v budove, kde sú prenosové rýchlosti medzi zariadeniami veľmi rýchle. Ďalšou možnosťou prepojenia výpočtových zariadení je rozľahlá sieť WAN, pomocou ktorej sme schopní prepojiť milióny počítačov po celom svete, avšak s pomalšími prenosovými rýchlosťami ako pri sieti typu LAN. Pomocou týchto inovácii môžeme veľmi ľahko vytvoriť výpočtový systém, ktorý je zložený z veľkého množstva zosieťovaných počítačov, ktoré sú väčšinou rozptýlené po celom svete. O takýchto počítačoch teda hovoríme, že tvoria distribuovaný systém. Distribuovaný systém môže tvoriť aj pár zariadení, ale väčšinou ide o väčší počet počitačov, ktoré sú prepojené buď káblom alebo bezdrôtovo, prípadne ich kombináciou. Ďalšou výhodou distribuovaných systémov je aj ich dynamika, čo znamená, že sa táto sieť môže neustále meniť a jej zariadenia po môžu pripájať a odpájať bez väčších následkov.[1]

2.1 Definícia distribuovaných systémov

Distribuované systémy by sme mohli charakterizovať nasledujúcou vetou: “Distribuovaný systém je zbierka autonómnych výpoč jednotiek, ktoré sa javia ich používateľom ako jeden súdržný systém.”[1] Túto charakteristiku môžeme rozdeliť na dve časti. V prvej časti sa pozrieme na to, že distribuovaný systém je zbierka samostatných výpočtových jednotiek, ktoré dokážu pracovať samostatne. Takéto výpočtové jednotky budeme nazývať uzly, a môžu to byť buď hardvérové zariadenia, alebo softvérové procesy. Pri dizajnovaní systémov sa nekladú žiadne požiadavky pre typ uzlov, takže to môžu byť zariadenia od veľkých počítačov, až po malé zariadenia v senzorových sieťach. V druhej časti si popíšeme to, že sa tieto jednotky javia ako jeden súdržný systém, čo znamená, že tieto jednotky musia medzi sebou komunikovať pomocou správ, čo je asi to najhlavnejšie pri dizajnovaní distribuovaných systémov.

Typ komunikácie medzi uzlami je tiež bez požiadaviek. Štruktúra distribuovaných systémov je zjednodušene zobrazená na obrázku 1.[1]



Obr. 1: Štruktúra distribuovaného systému[3]

2.1.1 Súbor samostatných výpočtových jednotiek

Moderné distribuované systémy sa väčšinou skladajú z rôznych druhov uzlov, od veľmi výkonných počítačov až po malé stolové počítače, prípadne ešte menšie zariadenia. Každý uzol môže konať nezávisle voči ostatným, pričom by sa nemali navzájom ignorovať, pretože by celý koncept distribuovaných systémov nemal význam. Funguje to tak, že uzly sa snažia dosiahnuť spoločný cieľ pomocou vzájomného vymieňania a zdielania informácií. Uzol reaguje na prichádzajúce požiadavky, spracuje ich a pošle na následné spracovanie do iných uzlov. Ďalšou vlastnosťou týchto uzlov je, že medzi nimi neexistuje niečo ako globálny čas, ale používa sa logický čas. To znamená, že každý uzol sa riadi podľa svojho vlastného času nezávisle na iných uzloch, čo je hlavne pre nepredvídateľné meškanie správ medzi nimi. Táto časová nesynchronizácia mieri na vnútornú synchronizáciu a koordináciu v rámci distribuovaného systému.[1]

Organizácia uzlov

To, že pracujeme s viacerými uzlami znamená, že sa musíme starať aj o ich členenie a organizáciu. Ide napríklad o separáciu uzlov, ktoré môžu, a ktoré nemôžu patriť do distribuovaného systému. Taktiež musíme poskytnúť každému uzlu zoznam uzlov, s ktorými môže komunikovať. Rozlišujeme dve varianty členenia uzlov:[1]

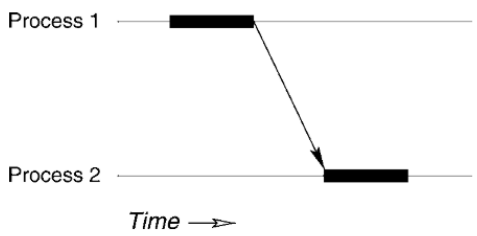
- Otvorené skupiny – Ktorýkoľvek uzol sa môže pridať k distribuovanému systému a môže posielať správy ktorémukoľvek uzlu v systéme
- Zatvorené skupiny – Môžu medzi sebou komunikovať iba jej členovia a pre prijímanie a odstraňovanie uzlov v skupine je nutný samostatný systém. Takýto systém overuje autenticitu uzlu, to ako komunikuje s ostatnými členmi, prípadne či nekomunikuje s inými uzlami mimo skupiny.

Distribuované systémy bývajú väčšinou organizované ako prekrytá sieť. V takom prípade by mala byť sieť vždy prepojená, čo znamená, že medzi každými dvoma uzlami by mala byť komunikačná cesta pre výmenu správ. Prepojenia uzlov bývajú často časovo náročné a zložité. Jedným zo známych a často používaný prepojení je pomocou P2P[1] sietí. Prekryté siete sú radené do dvoch tried[1]:

- *Štruktúrovaná prekrytá sieť* – Každý uzol má veľmi jasne definovaných susedov, s ktorými môže komunikovať. Uzly môžu byť organizované ako strom alebo v kruhovej topológii.
- *Neštruktúrovaná prekrytá sieť* – Každý uzol komunikuje s náhodne vybranými uzlami.

Výmena správ medzi uzlami

V prípade prekrytých sietí je uzol typicky softvérový proces so zoznamom procesov, ktorým môže posielať správy. Takéto procesy sú aktívne komponenty, ktoré majú stav a správanie. Stav sa skladá z dát, ktoré používa daný proces. Jeho správanie zodpovedá logike implementovanej v aplikácii. Ako už bolo naznačené, procesy medzi sebou komunikujú pomocou vymieňania správ. Tieto správy sa skladajú z postupnosti bajtov a sú medzi procesmi prenášané komunikačným médiom. Takúto komunikáciu môžeme vidieť v diagrame na obrázku 2, na ktorom je zobrazený prenos správy z Procesu 1 (odosielateľ) do Procesu 2 (prijímateľ) v čase.[3]



Obr. 2: Výmena správ medzi dvoma procesmi[3]

Tento diagram obsahuje časovú a stavovú závislosť procesov, pričom procesy môžu byť v aktívnom alebo pasívnom stave. V aktívnom stave sa na procese vykonávajú výpočty a je zobrazený hrubou čiarou na diagrame 2. Na tomto diagrame teda odosielateľ vykonáva výpočty a po odoslaní správy sa prepne do pasívneho stavu. Keď prijímateľ dostane správu, prepne sa do aktívneho stavu a začne vykonávať výpočty podľa obsahu správy.

[1] Peer-to-peer – architektúra distribuovanej siete

2.1.2 Jeden súdržný systém

Ďalšou časťou definície je, že distribuované systémy by sa mali javiť ako jeden súdržný systém. Iné definície hovoria aj o jednotnom systéme, čo znamená, že koncový užívateľ by nemal postrehnúť, že procesy a dáta sú rozmiestnené v rôznych uzloch a prebieha medzi nimi komunikácia a výmena dát. Dosiahnutie jednotného systému je však obtiažne a preto sa používa pojem súdržný systém. Pre vysvetlenie, v jednom súdržnom systéme sa súbor uzlov ako celok správa rovnako, pričom nezáleží kde, kedy a ako pôsobia užívateľ a systém medzi sebou. Môžeme teda povedať, že distribuovaný systém je súdržný, ak sa tak správa podľa očakávaní koncových užívateľov.[1]

Distribučná transparentnosť

Aj dosiahnutie súdržného systému je náročné, pretože sa napríklad požaduje, aby koncový užívateľ nemal prehľad, na ktorom uzle práve prebieha proces, prípadne či niektorá časť úlohy nebola preložená do iného procesu vykonávaného na inom uzle. Taktiež by nemal mať koncový užívatel prehlad o dátach, kde sú uložené, prípadne či nie sú vytvorené kópie dát pre zlepšenie výkonnosti systému. Takýto proces nazývame distribučná transparentnosť. Hlavnou výhodou distribučnej transparentnosti teda je, že nás už nezaujíma, kde sú zdroje uložené, vieme sa k ním dostať z ľubovoľného uzla v sieti a vytvára ilúziu, že celý systém je jeden počítač. Existuje viacero kritérií pre distribučnú transparentnosť, a to napríklad:[3]

- *transparentnosť umiestnenia* – Komponenty sú sprístupnené bez toho, aby používateľ vedel, kde sú fyzicky umiestnené.
- *transparentnosť prístupu* – Nezáleží, či sú komponenty umiestnené lokálne alebo na inom uzle, pretože prístup k ním bude rovnaký.
- *transparentnosť zlyhania* – Používatelia by nemali postrehnúť výpadok komponentu.
- *transparentnosť technológií* – Rozdiely v použitých technológiach sú skryté pred používateľmi, ako napríklad programovacie jazyky a operačné systémy.
- *transparentnosť súbežnosti* – Používatelia by nemali vedieť o tom, že zdieľajú komponenty s inými používateľmi.

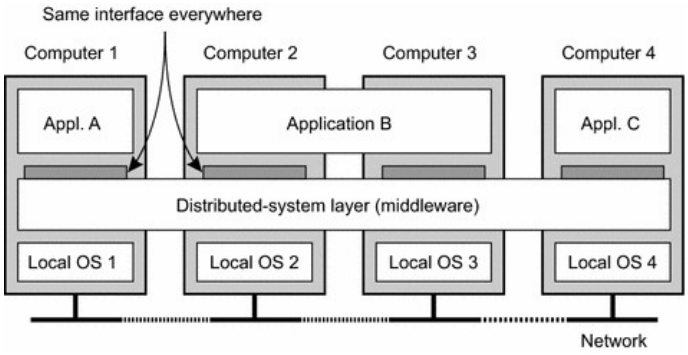
Príklad transparentnosti je podobný aj pri Unixových systémoch, v ktorých sa ku zdrojom pristupuje prostredníctvom zjednocujúceho rozhrania súborového systému, ktorý zakrýva rozdiely medzi súbormi, úložiskami, ale aj sieťami.[1]

Zlyhanie systému

Ďalší problém pri dosiahnutí jedného súdržného systému je ten, že systém pozostáva z viacerých prepojených uzlov, a hocikedy môže nastať výpadok ľubovoľného uzla alebo časti systému. Takýmto výpadkom môže nastať neočakávané správanie systému, napríklad sa niektoré procesy vykonajú správne a niektoré sa pozastavia. Čiastočné zlyhania sú prirodzené v každom komplexnom systéme, no v distribuovanom systéme je ich ťažko zakryť a je nutné tieto zlyhania odstrániť pre správne fungovanie systému.[1]

2.2 Middleware

Dôležitou súčasťou pri tvorbe distribuovaných systémov je middleware. Je to softvérová vrstva, ktorá sa stará o prepojenie operačného systému a aplikácií, aby zaistila kompatibilitu v komunikácii a výmene dát medzi uzlami systému. Táto vrstva je umiestnená nad operačným systémom každého uzlu systému, a ako napovedá aj názov, je medzi spomínaným operačným systémom a samotnými aplikáciami. Toto umiestenenie môžeme vidieť na obrázku 3. Tento obrázok zobrazuje štyri zosieťované počítače a tri aplikácie, pričom aplikácia B je distribuovaná medzi počítačom 2 a 3.[1]



Obr. 3: Middleware[1]

Middleware vrstva zabezpečuje, aby časti distribuovanej aplikácie komunikovali medzi sebou, ale tiež, aby bola umožnená komunikácia medzi rôznymi aplikáciami. Taktiež zakrýva rozdiely, ktoré sa vyskytujú v každom distribuovanom systéme. Tieto rozdiely sa môžu vyskytovať v týchto miestach[3]:

- **programovacie jazyky** – Aplikácie môžu byť vytvorené v rôznych programovacích jazykoch.
- **operačné systémy** – Aplikácie sú distribované na rôznych uzloch (zariadeniach), pričom uzly nemusia mať rovnaký operačný systém.

- **počítačové architektúry** – Uzly distribuovaného systému sa môžu skladať z rôznych výpočtových zariadení, ktoré sa môžu líšiť v technických detailoch(napr. reprezentácia dát).
- **siete** – Uzly môžu byť prepojené rôznymi druhmi sietí.

Zjednodušene by sa dalo povedať, že middleware je pre distribuovaný systém to isté, ako operačný systém pre počítač. Okrem toho, že middleware ponúka správu zdrojov po sieti, ponúka aj služby, ktoré môžeme nájsť v mnohých operačných systémoch (napr.: zariadenia pre komunikáciu medzi aplikáciami, bezpečnostné služby, správa účtov, maskovanie a zotavenie po poruchách). To, čím sa najviac odlišuje middleware od operačných systémov je, že jeho služby sú spojené so sieťou. Middleware zhromažďuje často používané časti aplikácií a funkcie, ktoré nemusia byť do aplikácií zvlášť implementované. Takéto middleware služby sú napríklad:[1]

- **komunikácia** – Pre komunikáciu medzi uzlami sa často používa RPC[2]. Ide o službu, ktorá dovoľuje aplikácii spustiť funkciu, ktorá je implementovaná a beží na vzdialenom počítači. Pre takéto vzdialené spustenie je nutný špeciálny programovací jazyk, ktorý umožní RCP službe vygenerovať potrebný kód pre nadviazanie vzdialeného spojenia.
- **transakcie** – Väčšina aplikácií je zložená z viacerých služieb, ktoré sú distribuované medzi uzlami systému. Middleware takéto služby spúšťa pomocou tzv. atomických transakcií. Ide o transakcie, kedy musia byť dostupné všetky požadované služby, a v prípade, že nejaké služby sú nedostupné, je celá transakcia medzi uzlami zamietnutá.
- **skladanie služieb** – Pri vývoji nových aplikácií je bežné, že sa použijú už existujúce programy a ich spojením vznikne nová aplikácia. V distribuovaných systémoch ide hlavne o webové aplikácie, a najmä také, ktoré sú webové služby. Middleware pomáha vytvárať cestu, akou sú sprístupnené služby, a poskytuje prostriedky pre vytváranie ich funkcií v konkrétnom poradí. Príkladom pre takéto skladanie služieb sú mashupy. Ide o webové stránky alebo aplikácie, ktoré používajú a kombinujú dáta z viacerých zdrojov (napr. Google Maps).
- **spoľahlivosť** – Pod týmto pojmom je myslená spoľahlivosť vo výmene správ medzi procesmi. Ak je aplikácia zložená z viacerých procesov a jeden z nich odošle správu inému, musí byť zaručené, že správu dostanú všetky procesy, alebo žiaden. Takéto funkcie výrazne zjednodušujú vývoj distribuovaných aplikácií a bývajú implementované ako časť middleware.

[2] Remote Procedure Call

2.3 Vlastnosti distribuovaných systémov

Uzly v distribuovanom systéme sa snažia dosiahnuť určitý spoločný cieľ. Pre dosiahnutie cieľa sa musia uzly riadiť a dizajnovať podľa určitých pravidiel, ktoré udávajú vlastnosti distribuovaných systémov. Okrem distribučnej transparentnosti popísanej v časti 2.1.2 patria medzi kľúčové vlastnosti distribuovaných systémov aj ďalšie, ktoré sú jednotlivo popísané nižšie.[4]

2.3.1 Zdieľanie zdrojov

Ako už bolo spomenuté, pri distribuovaných systémoch je dôležitá komunikácia medzi uzlami. Pre správne fungovanie takéhoto systému je potrebné, aby tieto uzly zdieľali medzi sebou svoje zdroje. Takéto zdroje môžu byť napríklad periférne zariadenia, úložiská, dáta, služby a rôzne iné. Je niekoľko dôvodov pre zdieľanie zdrojov, ale asi najhlavnejší je ekonomický. Ten môžeme vysvetliť napríklad na úložisku dát, kedy je výhodnejšie mať jedno výkonné úložisko zdieľané pre všetky uzly, ako keby mal každý uzol svoje vlastné úložisko a bolo by potrebné sa starať o každé zvlášť. Ďalším, asi najviac vystihujúcim príkladom pre zdielanie dát je P2P sieť BitTorrent, ktorá umožňuje efektívne zdieľanie dát medzi ľubovoľným počtom pripojených uzlov.

2.3.2 Otvorenosť

Otvorený distribuovaný systém je taký, kedy môžu byť všetky jeho komponenty ľahko dostupné, alebo integrované do iných systémov. Takýto systém sa tiež skladá z komponentov, ktoré pochádzajú z viacerých miest. Okrem hardvérovej stránky musí byť aj softvér v otvorenom distribuovanom systéme dizajnovaný tak, aby sa mohol vyvíjať a zdieľať medzi komponentami a inými systémami. Pre dosiahnutie otvorenosti by mali byť k dispozícii podrobné a dobre definované rozhrania komponentov, ktoré by mali byť štandardizované.

2.3.3 Súbežnosť

Súbežnosť je vlastnosť systému, pri ktorej môžu viaceré uzly vykonávať rovnakú funkciu v ten istý čas. Takéto súbežné vykonávanie aktivít môže prebiehať na rôznych komponentoch viacerých uzlov distribuovaného systému. Tiež môžu tieto funkcie vyvolať interakciu medzi uzlami. Súbežnosť má teda znížiť čakaciu dobu takýchto interakcií a zvýšiť výkon celého distribuovaného systému.

2.3.4 Škálovateľnosť

Škálovateľnosť sa zaoberá počtom uzlov, teda presnejšie, ako distribuovaný systém zvláda svoje zväčšovanie pri zvyšujúcom sa počte pridaných uzlov. Škálovateľnosť

teda zabezpečuje zvyšovanie výpočtu a spracovania v distribuovanom systéme. Najčastejšie sa distribuovaný systém škáluje pridávaním výpočtových uzlov do siete, pričom nemusíme meniť alebo upravovať staršie uzly systému. Uzly sa teda musia dizajnovať tak, aby mohli byť škálovateľné.

2.3.5 Odolnosť proti chybám

V distribuovaných systémoch je veľa vecí, napríklad hardvér, softvér alebo sieť, ktoré môžu z rôznych dôvodov prestať fungovať. Distribuovaný systém by mal čo najrýchlejšie detekovať a opraviť takéto zlyhania. Preto sa takéto systémy musia dizajnovať tak, aby boli dostupné aj pri výpadku rôznych komponentov.

3 Architektúra distribuovaných systémov

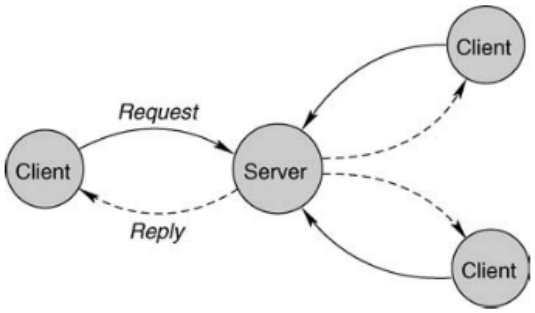
V tejto časti si priblížime možné architektúry distribuovaných systémov. Ide o organizáciu softvérových komponentov, miesto kde sa nachádzajú a ich vzájomné pôsobenie. Tieto architektúry môžeme rozdeliť na centralizované, decentralizované a rôzne hybridné formy.[1]

3.1 Centralizované organizácie

Pri centralizovaných organizáciách distribuovaného systému sa zavádzajú pojmy klient a server, pričom tieto pojmy lepšie vystihujú správanie procesov. Medzi centralizované organizácie patrí jednoduchá klient-server architektúra a viacúrovňové architektúry.[1]

3.1.1 Jednoduchá klient-server architektúra

Táto architektúra zavádza dve možné úlohy, ktoré môžu byť priradené procesu. Ide o úlohu používateľa služby, teda klienta, a o úlohu poskytovateľa služby, ktorú nazývame server. Server je proces, ktorý implementuje a poskytuje služby jednému alebo viacerým klientom, pričom môže ísť napríklad o databázovú službu. Klient je potom proces, ktorý požaduje službu zo servera tak, že mu posiela požiadavky a čaká na odpoveď. Tento kolobeh je zobrazený na obrázku 4.[1]



Obr. 4: Klient/Server architektúra[3]

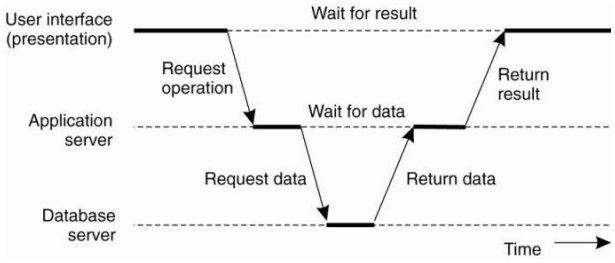
Jedným z hlavných komunikačných mechanizmov je už spomínaný RPC. Jeho iniciátorom je klient a server poskytuje vykonanie danej vzdialenej procedúry. Správa na obrázku 4, ktorú posiela klient na server, obsahuje všetky potrebné vstupné parametre pre vykonanie služby. Správa s odpoveďou zo servera potom obsahuje všetky výsledky z procedúry, ktoré požadoval klient.[3]

3.1.2 Viacúrovňové klient-server architektúry

V predchádzajúcom delení sme každému procesu priraďovali len jednu z jeho možných úloh, a to, že sa správal buď ako klient, alebo ako server. Môže sa však stať, že proces bude mať zároveň aj úlohu klienta aj servera. Táto situácia môže nastať, keď server bude potrebovať vykonanie procesu na inom serveri pre dokončenie svojho procesu. To môže nastať napríklad pri zapisovaní a čítaní dát z databázových serverov. Veľa distribuovaných aplikácií sa väčšinou delí na tri úrovne, a to:[1]

- používateľská vrstva
- výpočtová vrstva
- dátová vrstva

Príkladom pre takéto delenie môžu byť rôzne webové stránky a aplikácie. Na obrázku 5 sú zobrazené tieto tri úrovne, kde vidíme výmenu správ medzi nimi v čase. Prvá vrstva je používateľské rozhranie (klient), ktoré požaduje od aplikačného servera vykonanie operácie. Tento aplikačný server potom komunikuje s databázovým serverom. Ide o takzvanú vertikálnu distribúciu.[1]



Obr. 5: Príklad viacúrovňovej klient-server architektúry[1]

Možné zlyhania pri klient-server architektúre

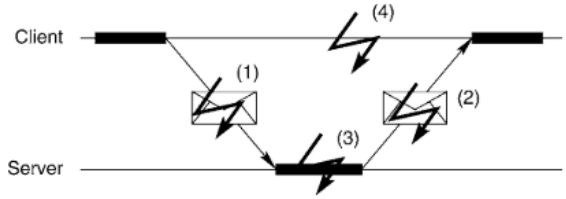
Pri volaní procedúr by nemal byť rozdiel v lokálnom volaní a vzdialenom volaní v distribuovaný systémoch. Distribúcia aplikácie však môže viesť ku viacerým zlyhaniam, ktoré by sa nemali vyskytovať v centralizovaných organizáciách. Tieto zlyhania môžu nastať buď pri strate správy, alebo pri zlyhaní serveru. To je graficky zobrazené na obrázku 6, a podľa očíslovania nasledovne:[3]

1. **strata vyžiadanej správy** – V prípade straty vyžiadavej správy klient opakovane zašle správu po určitom čase. Ten avšak nevie o akú chybu ide, a napríklad, ak ide o stratu výslednej správy, procedúra môže byť vykonaná dva krát. To môže nastať aj pri dlhom vykonávaní procedúry a čas pre opätovné zaslanie správy je príliš krátky.

2. **strata výslednej správy** – Ide o prípad, kedy sa vykonala procedúra na serveri, ale jej výsledná správa pre klienta sa stratila. Ako už bolo spomenuté pri predchádzajúcom zlyhaní, klient opätovne zašle správu. Ak server nezistí, že výsledná správa sa stratila, opätovne vykoná danú procedúru. To ale môže spôsobiť problém, ak procedúra pri predchádzajúcom vyvolaní zmenila stav servera.

3. **zlyhanie na strane servera** – Pri zlyhaní servera môže nastať situácia, kedy čiastočné vykonanie procedúry zmenilo stav servera. Takúto zmenu je nutné zistiť, pretože môže spôsobiť problém pri opätovnom vyvolaní procedúry. Ak sa napríklad zmenil obsah databázy, nie je jednoduché opätovné vyvolanie procedúry pre opravu a jej následné dokončenie.

4. **zlyhanie na strane klienta** – To môže nastať, ak zlyhá proces na strane klienta pri vykonávaní RPC. Vtedy server nevie, čo má s výsledkom spraviť a kam ho má poslať.



Obr. 6: Možné zlyhania[3]

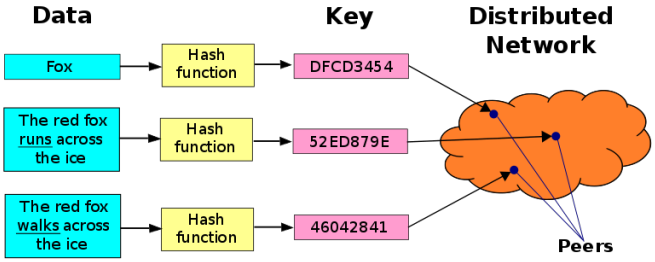
3.2 Decentralizované organizácie

Ďalšou z možných distribúcií klientov a serverov je horizontálna distribúcia. V tejto distribúcii môže byť klient alebo server fyzicky rozdelený na logicky rovnaké časti, pričom každá časť pracuje s vlastnými dátami, čím vyrovnáva záťaž. To znamená, že klient alebo server sú aj dodávatelia aj spotrebitelia zdrojov. Typom pre takúto architektúru, ktorá podporuje horizontálnu distribúciu, sú peer-to-peer (P2P) systémy.[1]

3.2.1 Štruktúrované peer-to-peer systémy

Pri štruktúrovaných P2P systémoch sa jednotlivé uzly skladajú do rôznych špecifických topológií, ako napríklad kruhová, stromová atď. Tieto topológie sú užitočné pre efektívne hľadanie dát. Charakteristickým znakom pre štruktúrované P2P systémy je indexovanie, to znamená, že dáta, s ktorými pracuje systém, majú špecifický kľúč, ktorý je použitý ako index. Ten sa väčšinou generuje pomocou hešovacích funkcií.

Takýto systém potom musí ukladať tieto kľúče a k nemu priraďujúce dáta, ktoré tvoria (kľúč,hodnota) páry, do takzvanej distribuovanej hešovacej tabuľky (DHT[1]). Podľa tejto tabuľky sa potom priraďuje vlastníctvo dát jednotlivým peerom a každý peer môže podľa nej hľadať a získať dáta od iných. Tento algoritmus je zobrazený na obrázku 7.[1]



Obr. 7: Algoritmus vytvorenia DHT[5]

3.2.2 Neštruktúrované peer-to-peer systémy

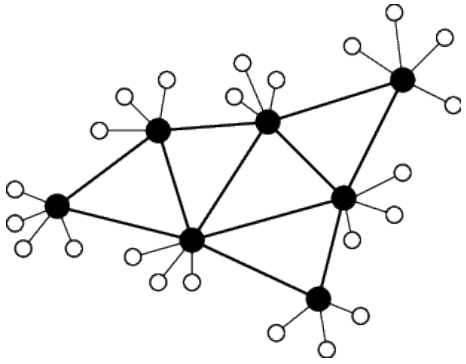
V neštruktúrovaných P2P systémoch si každý uzol spravuje ad hoc zoznam susedných uzlov. To vytvára výslednú štruktúru siete, ktorá sa nazýva náhodný graf. V takomto grafe existuje hrana medzi dvoma uzlami len s určitou pravdepodobnosťou. V ideálnom prípade je táto pravdepodobnosť pre všetky uzly rovnaká, ale v praxi sa vyskytujú rôzne distribúcie tejto pravdepodobnosti. Keď sa pripojí uzol do tohto systému, dostane štartovací zoznam ostatných uzlov v systéme, podľa ktorého môže nájsť ďalšie uzly, prípadne niektoré zamietnuť. Väčšinou sa tento zoznam neustále mení, napríklad, keď uzol zistí, že susedný uzol neodpovedá a musí ho nahradiť. Pre neustálu zmenu tohto zoznamu sa hľadanie dát v systéme nemôže vykonávať vopred definovanou cestou ako pri štruktúrovaných P2P systémoch, ale táto cesta sa neustále musí hľadať a meniť. Pre hľadanie dát existujú dve metódy, a to:[1]

- *flooding* – V tomto prípade pošle uzol požiadavku na získanie dát každému susednému uzlu. Ak sused nemá dáta, pošle požiadavku medzi svojich susedov. Táto požiadavka sa preposiela medzi uzlami, až dokým sa dané dáta nenájdu. Uzol, ktorý vlastní dáta, ich buď pošle priamo zadávateľovi požiadavky, alebo ich susedia postupne prepošlú k zadávateľovi.
- *random walk* – Ďalšou možnosťou je, že uzol pošle požiadavku na získanie dát náhodne vybranému susedovi. Ak uzol dáta nevlastní, prepošle požiadavku svojmu náhodne vybranému susedovi. Takto sa požiadavka preposiela dokým sa nenájde. Dáta sa následne pošlú ako v predchádzajúcom prípade. Táto metóda menej zaťažuje sieť, ale hľadanie dát trvá dlhšie. Pre urýchlenie sa môžu poslať požiadavky viacerým susedom naraz.

[1] DHT – distributed hash table

3.2.3 Hierarchicky organizované peer-to-peer systémy

Pri hierarchických P2P sieťach existujú dva druhy uzlov, a to "super peer" a "regular peer". Uzly super peer (čierny krúžok) sa používajú pre zhromažďovanie a správu dát z uzlov regular peer (biely krúžok) a tvoria P2P sieť. Takúto sieť môžeme vidieť na obrázku 8, v ktorej sa spája klient-server architektúra a P2P. Ako naznačuje obrázok, každý regular peer je pripojený k super peer ako klient, takže každá komunikácia s regular peer musí byť spracovaná daným super peer uzlom. Očakáva sa teda, že uzly super peer sú procesy s dlhou životnosťou a vysokou dostupnosťou. Ak sa ku sieti pripojí nový uzol, skoro vždy sa pripojí ako klient k nejakému super peer uzlu. V prípade, ak regular peer hľadá nejaké špecifické dáta, super peer rozpošle požiadavku medzi ostatné super peer uzly, ktoré mu následne dáta pošlú. Uzly regular peer sa tiež môžu pripojiť k inému super peer uzlu, ktorý vlastní vyhľadávané dáta.[1]



Obr. 8: Hierarchicky organizované peer-to-peer systémy[6]

3.3 Hybridné architektúry

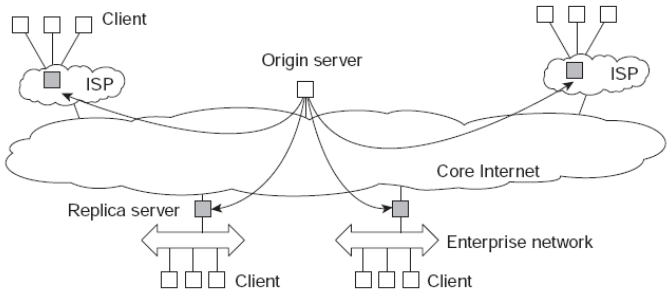
Veľa distribuovaných systémov kombinuje viacero architektúr, ako to je napríklad pri "super peer" sieťach. Aj hybridné architektúry vznikli spojením centralizovaných a decentralizovaných organizácií uzlov.

3.3.1 Edge-server systémy

Tieto systémy sa používajú v Internetovej sieti, v ktorej sa servery umiestňujú na okraj tejto siete. Okraj siete je tvorený hranicou medzi firemnými sietami a Internetom, ako to je pri poskytovateľoch internetových služieb (ISP[2]). Klienti sa teda pripájajú k Internetu pomocou okrajového servera, ako to je zobrazené na obrázku 9. Hlavným účelom okrajového servera je poskytovanie obsahu, ktorý môžu filtrovať, prípadne prepísať. Okrajové servery je možné aj spájať, kedy jeden server je ako hlavný server (origin server), z ktorého pochádza všetok obsah pre ostatné servery.

[2] Internet Service Provider

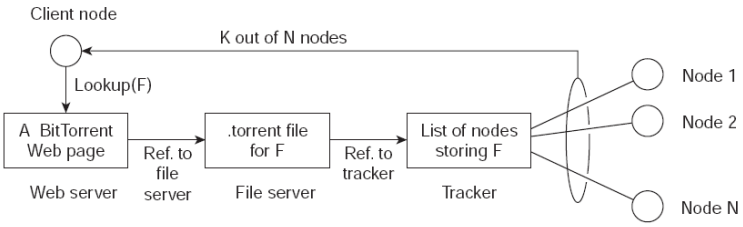
Hlavný server môže tiež používať ostatné okrajové servery pre replikáciu webových stránok a podobne.[1]



Obr. 9: Edge-server systémy[1]

3.3.2 Spolupracujúce distribuované systémy

Hybridné štruktúry sa používajú najmä pri spolupracujúcich distribuovaných systémoch. Pri štarte takéhoto systému, ktorý začínajú dva uzly, sa vytvorí klasická klient-server architektúra. Pri pripojení ďalších uzlov do systému je už možné používať plne decentralizovanú organizáciu uzlov. Takúto schému používa aj známy systém zdieľania súborov BitTorrent, ktorého schéma je zobrazená na obrázku 10. V tomto systéme si používatelia nájdu súbor, ktorý chcú stiahnuť, a po kúskoch ho stiahnu od ostatných používateľov. Tieto kúsky následne spoja do jednoho, čím vznikne sťahovaný súbor. Vo väčšine systémov zdieľania súborov používatelia len sťahujú súbory, a nezdieľajú ich ďalej. Práve kvoli tomuto zdieľaniu je v systéme BitTorrent možné sťahovať len vtedy, keď klient ktorý sťahuje poskytuje dáta ďalším klientom.[1]



Obr. 10: Fungovanie systému BitTorrent[1]

4 Typy distribuovaných systémov

Medzi tri základný typy distribuovaných systémov patria:[1]

- vysoko výkonné distribuované výpočty
- distribuované informačné systémy
- všadeprítomné systémy

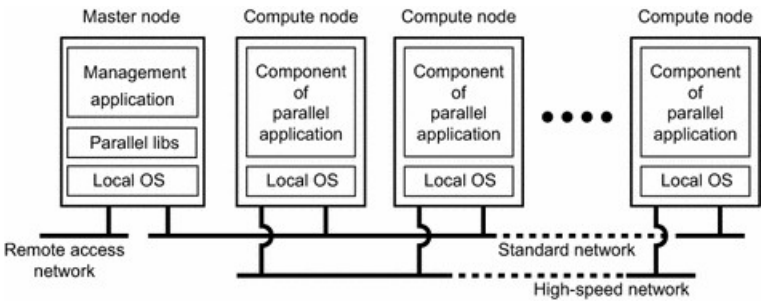
Tieto základné typy sa delia na ďalšie úrovne, ktoré si jednotlivo popíšeme v tejto kapitole. Pre potreby tejto práce sa najviac zameráme na prvý typ, pretože ide o typ distribuovaných systémov, ktoré budeme používať v implementácii modelu v praktickej časti.

4.1 Vysoko výkonné distribuované výpočty

Ide o dôležitý typ distribuovaných systémov, pretože práve tento typ umožňuje výpočet a používa sa pre náročné výpočtové úlohy. Pri tomto type existujú dve skupiny, akým sa dajú takéto výpočty spracovať v distribuovaných systémoch, a delia sa podľa hardvérových požiadaviek klientov. Prvá skupina sú klastrové výpočty, v ktorých hardvér musí byť zložený z rovnakých počítačov a musia mať rovnaký operačný systém. Tieto počítače musia byť spojené pomocou lokálnej siete. V prípade sieťových výpočtov môžu mať klienti rozličný hardvér a aj operačný systém. Tieto systémy väčšinou nie sú spojené pomocou lokálnej siete a väčšinou spadajú pod rozličné administratívne domény. Ďalšou možnosťou je použitie cloud výpočtov. V tomto prípade ide o poskytovanie výpočtových zdrojov.[1]

4.1.1 Klastrové výpočty

V dnešnej dobe sa zlepšuje pomer ceny a výkonu počítačov, čo zvýšilo popularitu klastrových výpočtových systémov. Existujú superpočítače, ktoré sú dostupné za vysokú cenu, ale od určitej hranice je výhodnejšie takýto superpočítač zostaviť pomocou spojenia viacerých jednoduchých počítačov prepojených pomocou vysokorýchlostnej siete. Klastrové výpočty sa väčšinou používajú pre paralelné programovanie, kedy sa jeden výpočtovo náročný program paralelne spúšťa na viacerých počítačoch. Často používaný príklad klastrového počítača tvoria linuxové Beowulf klastre. Takýto počítač tvorí skupina výpočtových uzlov, ktoré sú kontrolované a prístupné cez hlavný uzol, ako je zobrazené na obrázku 11. Hlavný uzol teda prideľuje uzly konkrétnym paralelným programom a poskytuje rozhranie pre používateľov. Hlavný uzol obsahuje middleware, ktorý je potrebný pre správu klastra, zatiaľ čo samotné výpočtové uzly obsahujú obyčajný operačný systém rozšírený o komunikačné prvky a podobne. Výpočtové uzly by mali byť identické, čo neplatí pre hlavný uzol.[1]



Obr. 11: Príklad klastrového počítača[1]

4.1.2 Sieťové výpočty

Na rozdiel od klastrových výpočtov sa pri sieťových výpočtoch neberie ohľad na homogenitu systému. To znamená, že pri systémoch sieťových výpočtov neexistujú predpoklady na podobnosť uzlov, takže uzly môžu mať rozličný hardvér, operačný systém, sieť a iné. Pri takýchto systémoch nastáva problém, že sa zhromažďujú údaje a zdroje z rôznych organizácii, aby sa umožnila ich spolupráca. Podstata sieťových výpočtov je teda efektívne spravovanie takýchto heterogénnych zdrojov, aby mohli byť plne využité pre výpočtové úlohy. Pre koncových užívateľov alebo samotné aplikácie sa takýto systém javí ako výkonný virtuálny počítač. Pre vytvorenie takéhoto systému je nutný špeciálny framework, ktorý zabezpečuje správu a pomocou ktorého sa vykonávajú výpočtové úlohy.[1]

4.1.3 Cloud výpočty

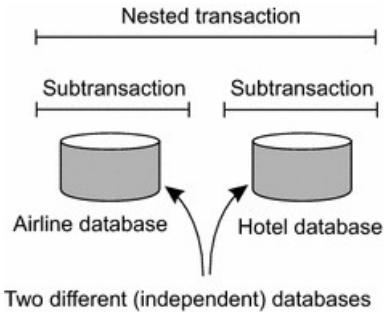
V prípade cloud výpočtov sa zdroje nespájajú ako pri sieťových výpočtoch, ale používateľom sú k dispozícii len zdroje na cloude. Odpadá teda problém so správou a integráciou viacerých zdrojov. Cloud výpočty teda v sebe zahŕňajú úložisko dát, a aj výpočtové zdroje. Poskytovatelia cloudu zabezpečia všetky zdroje a používatelia vidia systém ako jeden celok. Cloudové výpočty sa teda vyznačujú ľahko dostupnou a použiteľnou skupinou výpočtových zdrojov. Takéto výpočty sú väčšinou spoplatnené na základe zdrojov, ktoré boli poskytnuté používateľom pri výpočte úlohy. Zdroje sa dajú dynamicky nastavovať tak, že ak používateľ potrebuje vykonať viac práce, priplatí si viac zdrojov pre výpočet.[1]

4.2 Distribuované informačné systémy

Tento typ distribuovaných systémov používajú väčšinou organizácie, ktoré majú množstvo sieťových aplikácií, ktoré si medzi sebou vymieňajú množstvo dát. Tieto aplikácie sa integrujú do celofiremných informačných sytémov. Existujú dva druhy integrácie, ktoré si popíšeme nižšie.[1]

4.2.1 Distribuované transakcie

Pri tomto type sa zameráme hlavne na databázové aplikácie. Databázové operácie sa väčšinou vykonávajú pomocou transakcií. V distribuovaných systémoch sa transakcie väčšinou vytvárajú z podtransakcií, čo sú napríklad čiastočné transakcie pre viacero databáz, pričom vytvárajú takzvanú vnorenú transakciu. Takéto čiastočné transakcie môžu bežať paralelne na viacerých serveroch a databázach pre zvýšenie výkonu. Vnorené transakcie sú teda dôležité v distribuovaných systémoch, pretože zabezpečujú distribúciu transakcií na viacerých uzloch systému. Príkladom pre takúto transakciu môže byť plánovanie cesty, ktorá sa skladá z viacerých letov, a pre každý let sa spraví čiastočná transakcia. Každá z týchto čiastočných transakcií sa môže meniť nezávisle na ostatných. Príklad vnorenej transakcie je zobrazený na obrázku 12.[1]



Obr. 12: Príklad vnorenej transakcie[1]

4.2.2 Integrácia podnikových aplikácií

Pre oddeľovanie aplikácií od databáz, na ktorých boli postavené, sa museli aj podniky prispôsobiť tomuto trendu. Aplikácie by mali byť schopné vzájomne komunikovať, a priamo si vymieňať informácie. Pre komunikáciu sa dá použiť RPC metóda, ktorá zavolá funkciu na druhej aplikácii a vymenia si medzi sebou správy. Ďalšou možnosťou je použiť RMI[1] metódu, ktorá dokáže zavolať vzdialené objekty. Nevýhoda pri týchto metódach je tá, že obe aplikácie, medzi ktorými sa vymieňajú správy, musia byť počas komunikácie dostupné. Táto nevýhoda viedla k novej metóde zameranej na správy. Pri tejto metóde aplikácia posiela správy, pričom ostatné aplikácie môžu túto správu prijať, ak majú o ňu záujem. Takéto systémy sa nazývajú publish/subscribe systémy a tvoria dôležitú a rozširujúcu časť distribuovaných systémov.[1]

4.3 Všadeprítomné systémy

Mobilné a embedded výpočtové zariadenia tvoria systémy, ktoré sa nazývajú ako všadeprítomné. Ako napovedá názov, ide o systémy, ktoré sa snažia prirodzene ap-

[1] remote method invocations

likovať do prostredia. Sú to distribuované systémy, lebo spĺňajú ich vlastnosti. Všadeprítomné systémy sa skladajú z veľkého množstva senzorov, ktoré zachytávajú správanie používateľa. Tiež môžu obsahovať akčné jednotky, ktoré poskytujú spätnú väzbu zo senzorov. Väčšina zariadení v týchto systémoch je malá, mobilná, napájaná z batérie a komunikujúca len bezdrátovo, a ich úloha je v oblasti IoT[2].[1]

4.3.1 Všadeprítomné výpočtové systémy

Tieto systémy musia byť okrem všadeprítomnosti aj neustále dostupné. To znamená, že používateľ je neustále v kontakte s takýmto systémom, pričom si nemusí byť vedomý tohto kontaktu. Všadeprítomné výpočtové systémy by mali mať nasledovné požiadavky:[1]

- *distribúcia* – zariadenia sú zosieťované, distribuované a transparentné
- *interakcia* – pôsobenie medzi systémom a používateľom je nenápadné
- *súvislosť* – systém berie na vedomie kontext používateľa s cieľom optimalizovať vzájomné pôsosbenie
- *samospráva* – zariadenia pracujú bez zásahu používateľa, a sú samostatne riadené
- *inteligencia* – systém ako celok dokáže sám zvládať rôzne dynamické akcie a vzájomné pôsobenia

4.3.2 Mobilné výpočtové systémy

Pri týchto systémoch sa používa široká škála zariadení, ale najviac smartfóny a tablety. V poslednej dobe sa ale začali vyskytovať aj zariadenia ako diaľkové ovládače, vybavenie áut, rôzne GPS zariadenia atď. Všetky tieto zariadenia majú charakteristickú vlastnosť, a to, že používajú bezdrôtovú komunikáciu. Ďalšia vlastnosť je, že tieto zariadenia menia svoju polohu. Táto zmena môže spôsobovať problémy, pretože na každom mieste nemusia byť dostupné rovnaké služby. Preto sa používa dynamické objavovanie služieb. Tiež je nutné ohlasovanie polohy týchto zariadení, a preto potrebujeme poznať presné geografické súradnice.[1]

4.3.3 Senzorové siete

Senzorové siete sa skladajú z veľkého množstva senzorov. To, čo ich robí zaujímavým z hľadiska distribuovaných systémov je to, že sa tieto senzory snažia spolupracovať pre efektívne spracovanie snímaných údajov danej aplikácie. Ich správanie sa nelíši od normálnych počítačov, pretože tiež obsahujú softvérovú vrstvu, ktorá je nad hardvérovou, vrátane prístupu k sieťam, správy pamäte atď.[1]

[2] Internet of Things

5 Petriho siete

Petriho siete vynašiel v roku 1962 Carl Adam Petri, ktorý definoval formalizmus pre popis súbežných a synchrónnych distribuovaných systémov. Ide o modelovanie systémov, v ktorých sa vyskytujú synchronizačné, komunikačné a zdroje zdieľajúce procesy.[7]

5.1 Grafická reprezentácia

Na rozdiel od modelovania a formálneho analyzovania distribuovaných systémov sú Petriho siete vyjadrené grafickou reprezentáciou, ktorá sa vyznačuje svojou jednoduchosťou, prehľadnosťou a schopnosťou modelovania dynamiky procesov. Grafická reprezentácia takéhoto typu siete sa skladá z nasledujúcich komponentov:[7]

- *Miesta (Places)* – sú kreslené kruhmi ako je zobrazené na obrázku 13. Modelujú podmienky alebo objekty (premennú programu). Tieto miesta môžu obsahovať ľubovoľný počet tokenov, ale v špeciálnych prípadoch môžu byť obmedzené svojou kapacitou.

Obr. 13: Znak miesta

- *Tokeny (Tokens)* – Sú kreslené čiernymi bodkami ako je zobrazené na obrázku 14. Tokeny predstavujú špecifické hodnoty podmienok alebo objektov (hodnotu premennej programu).

Obr. 14: Znak tokenov

- *Prechody (Transitions)* – Sú kreslené obdĺžnikmi ako je zobrazené na obrázku 15. Pri ich aktivácii nastáva zmena hodnôt podmienok a objektov. Medzi dva hlavné typy prechodov patria okamžité a časované prechody.

(a) okamžitý prechod (b) časovaný prechod

Obr. 15: Znak prechodov

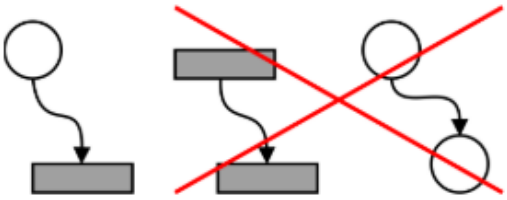
- *Orientované hrany (Arcs)* – Sú kreslené šípkami ako je na obrázku 16a, alebo špeciálnym typom hrany ako na obrázku 16b, ktorá obracia logiku vstupného miesta. Ide o vzájomné prepojenie miest a prechodov a určujú, ktoré objekty sa zmenia určitou aktivitou. Ich váhu (kapacitu) je možné meniť. Podľa hodnoty váhy sa odoberie také množstvo tokenov z miesta.

(a) hrana (b) inhibítor

Obr. 16: Znak hrany

5.1.1 Prepojenie komponentov

Petriho siete sú orientované biparitné grafy, v ktorých môžeme spojiť miesto a prechod, prípadne naopak, ale nie je možné spojiť dve miesta alebo prechody. Z tohoto vyplývjú aj skratky PN, napríklad P/T, čiže Place/Transitions, ktorá hovorí o spomínaných dvoch typoch uzlov použitých pri tejto sieti.[8]



Obr. 17: Povolené a zakázané prepojenia v P/T sieti[8]

5.2 Formálny popis Petriho siete

Orientovaný graf je formálne daný popisom jeho prvkov a funkciami alebo maticami, ktoré určujú ich vzájomné prepojenie. Formálne teda môžeme Petriho sieť definovať ako päticu $PN = (P, T, I^-, I^+, M_0)$, kde:[7]

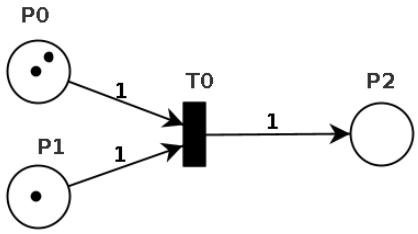
- $P = \{p_1, ..., p_n\}$ je konečná a neprázdna množina miest
- $T = \{t_1, ..., t_n\}$ je konečná a neprázdna množina prechodov
- $P \cap T = \emptyset$
- $I_- : P \times T \to \mathbb{N}_0$ je zpätná incidenčná funkcia
- $I_+ : P \times T \to \mathbb{N}_0$ je dopredná incidenčná funkcia
- $M_0 : P \to \mathbb{N}_0$ je počiatočné značenie

Funkcie I_- a I_+ popisujú spojenie medzi miestami a prechodmi. Ak je $I_-(p,t) > 0$, hrana je orientovaná od miesta p k prechodu t. Váha tejto hrany teda odpovedá množstvu tokenov, ktoré musí obsahovať miesto p pre umožnenie prechodu, a pri odpálení prechodu sa toľko tokenov z neho odoberie. Podobne je to aj pri funkcii $I_+(p,t)$, ktorá popisuje množstvo tokenov, ktoré sa pridajú do miesta p pri odpálení prechodu t.

5.3 Uskutočnenie prechodu v Petriho sieti

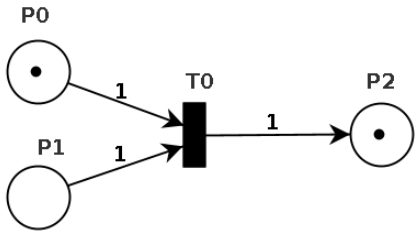
Uskutočnenie prechodu v Petriho sieti sa skladá z dvoch bodov, kde sa v prvom najskôr skontroluje umožnenie odpálenia prechodu a v druhom bode sa vykoná samotné odpálenie. Ich pravidlá sú nasledovné:[7]

- **Umožnenie prechodu** – Prechod je umožnený, ak obsahujú všetky vstupné miesta aspoň jeden token. V sieti na obrázku 18 je teda umožnený prechod *T0*. Po odpálení prechodu, čo je vidieť na obrázku 19, už nie je prechod *T0* umožnený pre nulový počet tokenov v mieste *P1*.



Obr. 18: Umožnenie prechodu

- **Odpálenie umožneného prechodu** – Umožnený prechod sa môže odpáliť. Pri tomto procese sa odoberie jeden token z každého vstupného miesta a pridá sa jeden token na každé výstupné miesto prechodu. To môžeme pozorovať na rozdieloch medzi obrázkom 18 a obrázkom 19, kedy sa z miest *P0* a *P1* odoberie jeden token a pridá s do miesta *P2*.



Obr. 19: Odpálený prechod

5.3.1 Prioritné prechody

Ak existuje viacero prechodov, ktoré sú umožnené z jedného miesta, môžeme dať každému prechodu prioritu $p_t \in \{0, 1, 2, 3, ...\}$. Väčšie číslo znamená vyššiu prioritu. Prechody s prioritou sú povolené, ak sú umožnené v obyčajnej Petriho sieti bez priorít, a žiadne iné prechody nemajú vyššiu prioritu. Ak je takýto prechod odpálený, zmení sa stav siete ako pri sieti bez priorít.[9]

5.3.2 Konfliktné prechody

Ide o situáciu, kedy je v aktuálnom stave umožnených viac prechodov. Ak sa tieto prechody navzájom neovplyvňujú, môžu byť odpálene v ľubovoľnom poradí. Takéto prechody nazývame nezávislé. V opačnom prípade, teda ak jeden prechod spôsobí, že iný prestane byť umožnený, môže sa systém dostať do niekoľko rôznych stavov. Vtedy hovoríme o konfliktných prechodoch. Konfliktné prechody modelujú súperenie o zdroje a vzájomnú výlučnosť viacerých udalostí. V prípade nezávislých prechodov ide o modelovanie asynchrónnosti a paralelizmu.[8]

5.4 Typy Petriho sietí

Poznáme niekoľko typov Petriho sietí, ktoré sa vyznačujú svojimi vlastnosťami, a každé majú svoje použitie. Medzi najbežnejšie patria obyčajné Petriho siete, ale pre potreby tejto práce sa budeme najviac zaoberať časovanými Petriho sietami, ktoré majú významnú rolu pri modelovaní stochastických systémov. Typy Petriho sietí môžeme rozdeliť nasledovne:[9]

1. **Obyčajné Petriho siete**
    - *C/E (Condition/Event) Petriho siete* – Ide o najjednoduchší typ Petriho sietí. Miesta môžu nadobúdať boolovské hodnoty, to znamená, že miesto môže obsahovať najviac jeden token.
    - *P/T (Place/Transitions) Petriho siete* – Miesta môžu nadobúdať celočíselné hodnoty, to znamená, že miesto môže obsahovať množstvo tokenov. Tento druh sietí môže mať ešte nasledovné modifikácie:
        - *P/T Petriho siete s inhibičnými hranami* – Ide o špeciálny prípad hrán, ktoré sa používajú pre opačnú logiku vstupného miesta. To znamená, že absencia tokenu na vstupnom mieste aktivuje prechod.
        - *P/T Petriho siete s prioritami* – Ide o modifikáciu, kedy každý prechod môže mať inú prioritu pre jeho uskutočnenie.

2. **Petriho siete vyššej úrovne**

   - *farebné Petriho siete (CPN[1])* – Pri týchto sieťach môžu mať tokeny rôzne vlastnosti (hodnoty), ktoré vyjadruje ich farba, čím sa ovplyvní ich výskyt v miestach.
   - *hierarchické Petriho siete (HPN[2])* – Umožňujú dekompozíciu systému na menšie časti, ktoré sa modelujú samostatne a spoločne vytvoria celkový model siete.
   - *objektovo orientované Petriho siete (OOPN[3])* – Tieto siete umožňujú úplnú integráciu objektovo orientovaného konceptu, vrátane dedičnosti, polymorfizmu a dynamickej väzby.

3. **Časované Petriho siete**

   - *časované Petriho siete (TPN[4])* – Pri tomto type Petriho sietí môžeme modelovať čas, kedy sa daný prechod aktivuje, čo je dôležitý faktor pri modelovaní stochastických systémov. Tento druh siete budeme využívať v praktickej časti tejto práce.

[1] CPN – Coloured Petri nets
[2] HPN – Hierarchical Petri nets
[3] OOPN – Object Oriented Petri nets
[4] TPN – Timed Petri nets

6 Časované Petriho siete

Pomocou základných Petriho sietí, napríklad P/T sietí, nie je vhodné analyzovať stochastické systémy, pretože nezahŕňajú časovú informáciu o tom, kedy sa odpáli prechod. Takéto siete sú vhodné pre kvalitatívnu analýzu, to znamená analyzovanie funkčného alebo kvalitatívneho správania systému. Zavedením času do Petriho sietí zásadne zmeníme ich správanie a umožníme ich analyzovať aj kvantitatívne. Zavedenie času do Petriho sietí dosiahneme dvoma základnými spôsobmi, ktoré sú spojené s komponentami siete nasledovne:[7]

- **definovaním času pobytu tokenov v miestach** – Tokeny, ktoré sa odpáleným prechodom dostali do miesta p, ostanú nedostupné pre všetky nasledujúce prechody po stanovenú dobu. Po uplynutí tejto doby sa tokeny z miesta p opäť stanú dostupnými pre výstupné prechody tohoto miesta. Takéto siete nazývame $TPPN$[1].
- **definovaním času omeškania odpálenia prechodu** – Po umožnení prechodu nastane odpálenie až po určitej definovanej dobe. Takéto siete sa nazývajú $TTPN$[2], a môžeme ich rozdeliť do dvoch skupín:
  - *modely s predbežným výberom* – Po umožnení prechodu t sa všetky tokeny rezervujú pre t, čím zabránia, aby boli dostupné pre iné prechody.
  - *závodné modely* – V tomto prípade nie sú tokeny z miesta p rezervované prechodom t, a sú dostupné pre všetky prechody z p. Po uplynutí definovanej doby sa prechod t odpáli, ak bol aj po tejto dobe umožnený. To znamená, že všetky umožnené prechody z p súťažia o jeho tokeny, a rýchlejšie prechody znemožnia odpálenie ostatných.

Ďalšie možnosti zavedenia času do Petriho sietí sú úzko spojené s predchádzajúcimi dvoma možnosťami. Jedná sa o variácie doby trvania prechodu, ktoré sú aplikované na ostatné stavebné prvky siete, a to nasledovne:[9]

- **definovaním času pobytu tokenov na hrane** – V tomto prípade si môžeme predstaviť, že tokeny sa môžu pohybovať po hranách určitou konečnou rýchlosťou. To znamená, že doba cesty od vstupných miest k prechodu sa rovná trvaniu prechodu.
- **definovaním časového razítka (time stamp) tokenov** – Predpokladáme, že vykonanie prechodu je okamžité, ale pri opustení tokenov z prechodu dostanú tokeny časové razítko. Toto razítko zaznamenáva aktuálny globálny čas,

[1] TPPN – Timed Places Petri nets
[2] TPPN – Timed Transitions Petri nets

ktorý je v momente vykonania prechodu zvýšený o stanovenú dobu trvania prechodu. Tokeny s časovým razítkom môžu byť znova použité až po uplynutí tohoto času.

Doby odpálenia prechodov pri časovaných Petriho sieťach môžu byť charakterizované jednou z nasledujúcich možností:[9]

- **deterministicky** – Stanovená doba odpálenia je konštanta. V tomto prípade hovoríme o časovaných Petriho sieťach (TPN)
- **stochasticky** – Stanovená doba odpálenia prechodu je náhodná, väčšinou s exponenciálnym rozložením. Pri tomto prípade ide o stochastické Petriho siete (SPN[3]).
- **kombináciou predchádzajúcich spôsobov** – Stanovené doby sú pre niektoré prechody konštanty, pre iné náhodné veličiny. V tomto prípade hovoríme o zobecnených stochastických Petriho sieťach (GSPN[4]).

6.1 Stochastické Petriho siete

Stochastické Petriho siete sú definované ako $SPN = (PN, \Lambda)$, kde PN je definícia základných Petriho sietí, ku ktorej pridáme exponenciálnu náhodnú veličinu $\Lambda = (\lambda_1, ..., \lambda_m)$, priradenú ku každému prechodu $T = (t_1, ..., t_m)$. Platí teda, že všetky vlastnosti pôvodnej základnej PN dedí aj z nej vytvorená stochastická PN.[7] Distribučná funkcia exponenciálnej náhodnej veličiny χ_i s dobou prechodu t_i je potom[7]

$$F_{\chi_i}(t_i) = 1 - e^{-\lambda_i t_i}$$

Priemerná doba odpálenia prechodu t_i je nepriamo úmerná parametru λ, čo vyjadríme ako[7]

$$T = \frac{1}{\lambda}$$

6.2 Zobecnené stochastické Petriho siete

Z hľadiska distribuovaných systémov ide o najzaujímavejší prípad Petriho sietí. Stochastické Petriho siete sú tiež vhodné pre modelovanie reálnych systémov, ale kvantitatívna analýza stabilizovaného stavu je náročná pre veľký rozptyl hodnôt intenzít prechodov. Preto sa ku stochastickým PN pridáva aj ďalší typ prechodov, a skladajú sa teda z:[9]

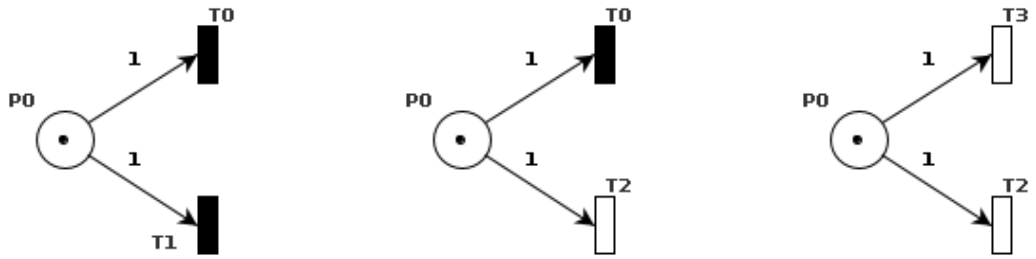
[3] SPN – Stochastic Petri nets
[4] GSPN – Generalized Stochastic Petri nets

- **okamžitých prechodov** – Ak je prechod umožnený, je odpálený okamžite, teda v nulovom čase.
- **časovaných prechodov** – Tieto prechody sú odpálené až po náhodnom, exponenciálne distribuovanom čase ako v prípade SPN.

6.2.1 Konfliktné prechody

Konfliktné prechody sa riešia podobne ako pri základných Petriho sieťach. Ide teda o vzájomné vylúčenie viacerých umožnených prechodov. V tomto prípade máme dva typy prechodov, preto definujeme konfliktné chovanie pre ich vzájomnú kombináciu. Každý prechod bude mať priradenú nejakú váhu (pravdepodobnosť). Možné kombinácie sú zobrazené na obrázku 20, a sú to:[9]

- **Konfliktné okamžité prechody** – Ku každému z konfliktných prechodov je priradená pravdepodobnosť, s akou nastane jeho odpálenie.
- **Konfliktný časovaný a okamžitý prechod** – Vždy sa odpáli ako prvý okamžitý prechod, ak je umožnený.
- **Konfliktné časované prechody** – Ak majú tieto prechody parametre λ_i, môžeme vypočítať pravdepodobnosť, že bude odpálený prechod k s parametrom λ_k, a to podielom $\frac{\lambda_k}{\sum_i \lambda_i}$.



(a) okamžité prechody (b) kombinácia prechodov (c) časované prechody

Obr. 20: Konfliktné prechody GSPN

7 Modelovanie distribuovaných systémov

Pre modelovanie distribuovaných systémov existuje viacero softvérov a knižníc v rôznych jazykoch. Softvérové riešenia väčšinou neposkytujú dostatok prostriedkov pre vytvorenie Petriho sietí väčších rozmerov. V prípade knižníc je možné tvoriť aj väčšie siete, ktoré sa dajú jednoducho naklonovať, buď implementáciou v knižnici, alebo programovo v jednotlivých jazykoch. Tieto softvéry a knižnice však nemusia poskytovať všetky typy Petriho sietí, a tak môžu byť vhodné len pre niektoré aplikácie. Pre implementáciu modelov v praktickej časti tejto práce bola zvolená knižnica *PetNetSim*[2], ktorá je napísaná v jazyku *Python*. Knižnica *PetNetSim* bola vyvinutá na ústave Automatizácie a informatiky našej školy. Táto knižnica poskytuje všetky potrebné typy Petriho sietí, ktoré sú nutné pre implementáciu modelov.

7.1 Knižnica PetNetSim

Knižnica *PetNetSim* ponúka okrem knižnice aj grafický editor, ktorý komunikuje s knižnicou pomocou súborového formátu *json*[1]. Vytvorené siete v editore je možné uložiť do tohto formátu, a následne pomocou knižnice upravovať. Tento postup je možný aj v opačnom poradí. V tejto knižnici sú implementované obyčajné Petriho siete, a tiež aj časované Petriho siete. Ako už bolo spomenuté, ide o knižnicu napísanú v jazyku *Python*, ktorá využíva objektovo orientované progamovanie. Všetky triedy objektov majú parameter *name*, ktorý je nastavený na počiatočnú hodnotu *None*, a priradí sa im nejaké meno. Pokial chceme zadať vlastné meno, musíme nahradiť hodnotu *None* vlastným textovým reťazcom. Triedy obejktov sú definované nasledovne:[2]

- **Place** – Táto trieda reprezentuje miesto v Petriho sieti. Miesto môže mať počiatočný počet tokenov zadaných parametrom *init_tokens*. Tiež môže mať obmedzenú, alebo neobmedzenú kapacitu, ktorá je zadaná parametrom *capacity*. Neobmedzená kapacita je daná premennou *INF_CAPACITY*.

```
1 Place(name=None, init_tokens=0, capacity=INF_CAPACITY)
```

- **Transition** – Ide o reprezentáciu obyčajného prechodu, ktorý nemá špeciálne vlastnosti. Z tejto triedy sa dedia nasledujúce prechody:
  - **TransitionPriority** – Ide o prechod s nejakou prioritou. Táto priorita je zadaná parametrom *priority*.

```
1 TransitionPriority(name=None, priority)
```

[1] formát pre prenos dát, ktoré môžu byť organizované v poliach alebo agregované v objektoch

  - **TransitionTimed** – Táto trieda reprezentuje časované prechody. Čas vykonania prechodu môže byť konštantný, kedy sa berie čas *t_max*, alebo sa pomocou distribúcie zadanej parametrom *p_distribution_func* vyberie hodnota z jeho parametrov *t_min* a *t_max*.

```
TransitionTimed(name=None, t_min, t_max=1,
p_distribution_func=constant_distribution)
```

  - **TransitionStochastic** – Ide o prechod s pravdepodobnosťou, s akou bude odpálený. Táto pravdepodobnosť je daná parametrom *probability*. Tento prechod nie je možné kombinovať s inými prechodmi v konfliktných skupinách.

```
TransitionStochastic(name=None, probability)
```

- **Arc** – Táto trieda reprezentuje hranu, ktorá môže ísť z miesta do prechodu, alebo naopak. Hrana začína z parametra *source* a ide do *target*. Tieto parametre uvádzajú názov miesta alebo prechodu. Hrana môže brať určitý počet tokenov, ktorý je zadaný parametrom *n_tokens*.

```
Arc(source, target, n_tokens=1, name=None)
```

- **Inhibitor** – Ide o samostatnú triedu, ktorá reprezentuje inhibičnú hranu. Jej parametre sú rovnaké ako pri triede **Arc**.

```
Inhibitor(source, target, n_tokens=1, name=None)
```

Výsledná sieť je reprezentovaná ako objekt triedy **PetriNet**. Tento objekt vytvoríme spojením všetkých zadefinovaných miest, prechodov a hrán. Pre časované prechody obsahuje premennú **time**.[2]

```
PetriNet(places, transitions, arcs)
```

Táto trieda má viacero metód, ktoré sú nutné pre ďalšie spracovanie siete a jej simuláciu. Sú to metódy:[2]

- **clone** – Táto metóda zabezpečí klonovanie vytvorenej siete. Ku každému názvu objektu v naklonovanej sieti sa pre rozlíšenie pridá predpona, ktorá je daná parametrom *prefix*. Do novovytvorenej siete obsahujúcej naklonované podsiete môžeme pridať ďalšie miesta, prechody a hrany dané parametrami *places*, *transitions* a *arcs*.

```
PetriNet.clone(prefix: str, places, transitions, arcs)
```

- **step** – Táto metóda zabezpečí odpálenie prechodov. Tie sa odpaľujú na základe konfliktných skupín. Táto metóda neberie žiadne parametre. Pre vykonanie celej simulácie, teda vykonanie premiestnenia tokenov do konečných miest, musíme túto metódu vykonávať v cykle.

```
while not petri_net.ended: PetriNet.step()
```

- **reset** – Ide o metódu, ktorá zabezpečí obnovenie siete do počiatočného stavu, a napríklad nastavenie premennej **time** na hodnotu 0. Neberie žiadne parametre.

```
PetriNet.reset()
```

7.1.1 Konfliktné skupiny

Knižnica *PetNetSim* rieši konfliktné prechody tak, že si najskôr dopredu vypočíta, ktoré prechody môžu byť v konflikte. Všetky možné konfliktné prechody sa potom v implementácii nazývajú ako konfliktná skupina.

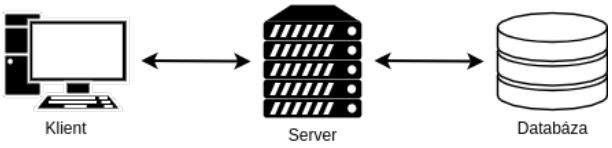
8 Implementácia 1: Horizontálne škálovaná webová aplikácia rozdelená na viac služieb s distribuovanou databázou

V tejto kapitole popíšem vytvorený model distribuovanej webovej aplikácie. Najskôr je ale vhodné priblížiť obyčajnú architektúru webovej aplikácie a jej škálovanie, mikroslužbovú architektúru a delenie požiadaviek medzi mikroslužbami. Pomocou diagramov a informácií z týchto architektúr som zostavil Petriho sieť distribuovanej webovej aplikácie.

8.1 Architektúra monolitickej webovej aplikácie

Základný princíp fungovania monolitickej webovej aplikácie môžeme znázorniť na typickej architektúre, v ktorej zatiaľ neuvažujeme o škálovaní. Ide o viacúrovňovú klient-server architektúru, ktorá je podrobnejšie popísaná v časti 3.1.2. Táto architektúra je zobrazená na diagrame 21, na ktorom sú tri komponenty, ktoré sú medzi sebou prepojené, a to:

- **Klient** – posiela svoje požiadavky na server
- **Server** – spracováva prijaté požiadavky od klientov a komunikuje s databázou
- **Databáza** – slúži ako úložisko dát pre výpočty na serveri



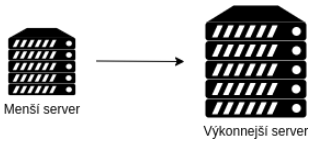
Obr. 21: Architektúra webovej aplikácie

8.1.1 Škálovanie webovej aplikácie

S narastajúcim počtom záujemcov o webové aplikácie a stránky je nutné, aby ustáli nápor tisícok používateľov bez straty výkonu, a taktiež, aby boli odolné voči výpadkom a chybám. To docielime tým, že danú webovú aplikáciu škálujeme. Existujú dva typy škálovania, a to:[10]

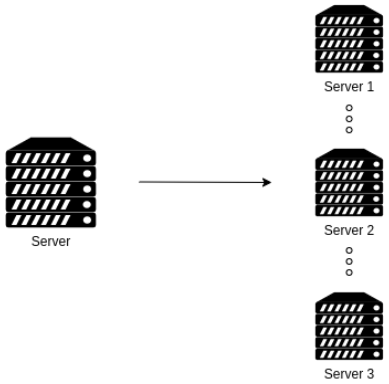
- **Vertikálne škálovanie** – Ide o pridanie zdrojov do existujúceho servera, prípadne o nahradenie servera výkonnejším. Jeho architektúra ostáva rovnaká, len sa zväčšuje jeho výkon, aby vyhovoval väčšiemu počtu klientov. Takéto vylepšenie je však len dočasné, pretože s narastajúcim počtom klientov už nemusia byť pridané zdroje dostačujúce, a tak je nutné znova pridať ďalšie

zdroje. Ďalšou nevýhodou pri vertikálnom škálovaní je aj to, že pri zlyhaní jednej časti systému zlyhá celý systém. Vertikálne škálovanie je zobrazené na diagrame v obrázku 22.



Obr. 22: Vertikálne škálovanie

- **Horizontálne škálovanie** – Ide o pridenie ďalších serverov, ktoré slúžia na ten istý účel. Požiadavky sa v takomto systéme rozdelia rovnomerne medzi všetky servery, a tak nie je zaťažovaný len jeden server, a vybavenie požiadaviek je rýchlejšie. Tento druh škálovania je efektívnejší oproti vertikálnemu škálovaniu, pretože je veľmi jednoduché do systému pridať ďalší server, namiesto toho, aby sme jeden server vylepšovali. Tento druh škálovania je použitý aj pri implementácii modelu distribuovanej aplikácie. Horizontálne škálovanie je zobrazené na diagrame v obrázku 23.



Obr. 23: Horizontálne škálovanie

Loadbalancing

Pre rovnomerné rozdelenie požiadaviek od klientov medzi servermi sa používajú loadbalancery. Tie slúžia ako sprostredkovateľ medzi klientami a servermi, a smerujú jednotlivé požiadavky klientov medzi servery. Existuje viacero metód, ktorými sa load balancery riadia, ale najviac používaná a ľahko implementovateľná je metóda **round robin**, ktorá cyklicky posiela požiadavky na servery. Túto metódu je možné vysvetliť na nasledujúcom príklade. Ak máme dva servery čakajúce na požiadavky, tak pri príchode prvej požiadavky ju bude load balancer smerovať na prvý server. Druhú požiadavku bude load balancer smerovať na druhý server. Keďže máme len dva servery, tak tretia požiadavka bude smerovaná na prvý server. Táto metóda je najúčinnejšia, ak sú všetky servery rovnako výkonné. Problém môže nastať, ak

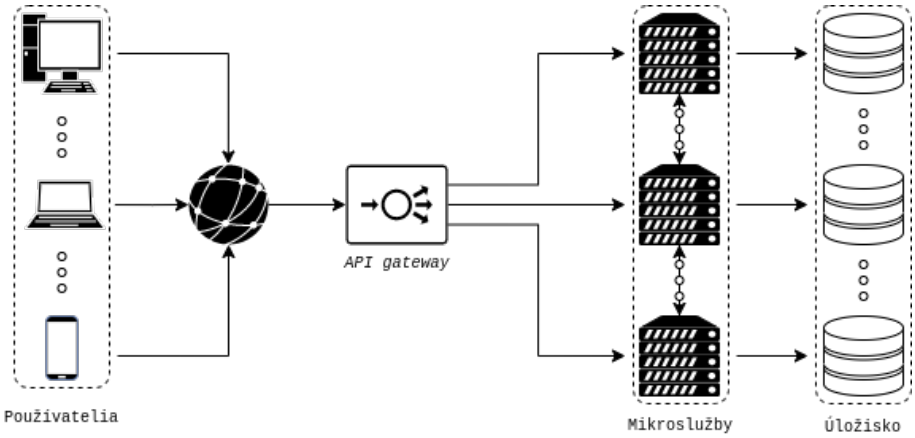
majú servery rôzne parametre, a jeden server sa môže preťažiť skôr ako ostatné. Metóda round robin teda neberie ohľad na parametre serverov, tiež neberie ohľad na výpočtovú náročnosť požiadaviek, a stále smeruje požiadavky rovnomerne.[11]

8.2 Mikroslužbová architektúra

Ďalej je vhodné si priblížiť mikroslužbovú architektúru. Tento pojem popisuje spôsob navrhovania aplikácií, ako súbor služieb, ktoré sa môžu vyvíjať nezávisle na sebe. Monolitická aplikácia je teda rozdelená na viacero menších služieb, ktoré pracujú ako samostatné procesy a komunikujú medzi sebou pomocou API brány. Tieto služby môžu byť naprogramované v rôznych jazykoch a tiež môžu používať rôzne technológie pre ukladanie údajov. Mikroslužby sú väčšinou jednoúčelové, a vykonávajú len jeden typ úlohy z celej aplikácie. Často bývajú mikroslužby horizontálne škálované, a to podľa využitia komponentov systému.[12]

8.2.1 API brána

Keď už vieme, ako vyzerá architektúra webovej aplikácie s jedným serverom a databázou, môžeme aplikáciu rozdeliť na mikroslužby, čím nám vznikne schéma, ktorá je zobrazená na diagrame v obrázku 24. Na tejto schéme môžeme vidieť okrem základných komponentov, ktoré sú obsiahnuté aj v jednoduchej architektúre, aj API bránu. Ide o dôležitý komponent, ktorý slúži pre distribuovanie požiadaviek medzi mikroslužbami. Všetky požiadavky najskôr idú cez API bránu, ktorá ich následne smeruje na požadovanú mikroslužbu. Príklad takéhoto smerovania môžeme ukázať na aplikácii internetového obchodu, kde mikroslužby môžu byť napríklad produktové informácie, nákupný košík, hodnotenia produktov a podobne. Ak má teda klient požiadavku na informácie o produkte, API brána bude túto požiadavku smerovať na mikroslužbu s produktovými informáciami.[13]

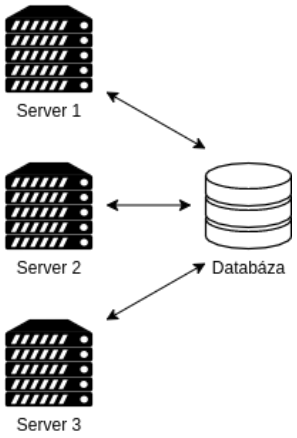


Obr. 24: Model distribuovanej webovej aplikácie

8.2.2 Databáza distribuovanej aplikácie

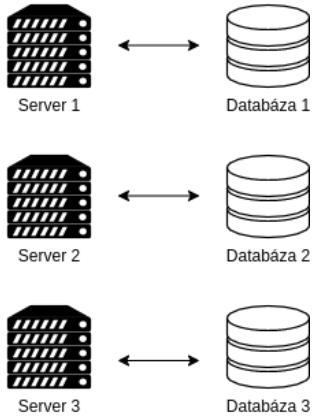
Ďalšou dôležitou vecou, ktorá súvisí so škálovaním, je aj rozloženie monolitickej databázy. Namiesto jednej databázy je výhodné použiť viacero menších databáz, ktoré budú súvisieť s každou časťou aplikácie zvlášť. To znamená, že každá mikroslužba bude mať vlastnú databázu, ktorá bude obsahovať len dáta, ktoré používa. To umožní, aby sa mohli mikroslužby nezávisle vyvíjať a ďalej škálovať. Existujú dva spôsoby, ako môže fungovať správa dát v distribuovanej aplikácii, a to:[14]

- **monolitická databáza** – Tradičné aplikácie majú väčšinou jednu zdieľanú databázu, v ktorej sú dáta zdieľané medzi viaceré časti systému, teda medzi mikroslužby. Takáto databáza má výhodu, že všetky dáta sú na jednom mieste, a jej vývoj je jednoduchší. Monolitická databáza je zobrazená na diagrame v obrázku 25. Avšak má viacero nevýhod, a to napríklad:[14]
  - *nemožnosť nezávislého vývoja mikroslužieb* – ak k databáze pristupujú viaceré mikroslužby, je nutné po každej zmene v databéze aktualizoavť aj každú mikroslužbu
  - *škálovanie jednotlivých mikroslužieb* – je ťažké škálovať aplikáciu s monolitickou databázou, pretože je len jedna možnosť, to a škálovať databázu vertikálne, čiže zväčšovať jej objem
  - *spomalenie aplikácie* - po určitom čase sa môže v monolitickej databáze nahromadiť veľa dát, čím je ich vyhľadávanie a práca s nimi pomalšia
  - *obsahuje väčšinou relačnú databázu* – to znamená, že všetky mikroslužby musia pracovať s relačnou databázou, ktorá nemusí byť optimálna pre všetky mikroslužby.



Obr. 25: Diagram monolitickej databázy

- **distribuovaná databáza** – Ide o prípad, kedy má každá mikroslužba svoju vlastnú databázu, ktorá obsahuje dáta spojené s danou mikroslužbou. To umožní, aby sa mohli jednotlivé služby nezávisle na sebe vyvíjať. Distribuovaná databáza je zobrazená na diagrame v obrázku 26.



Obr. 26: Diagram distribuovanej databázy

Pri vytváraní distribuovanej databázy by sme mali brať ohľad na to, že každá mikroslužba bude mať vlastný pohľad na systém. To znamená, že každá mikroslužba má svoje dáta, a nemala by vidieť do dát inej mikroslužby. Databázy by mali byť súkromné pre každú mikroslužbu, a žiadna iná mikroslužba by nemala priamo meniť dáta nejakej inej mikroslužby. Ďalšou možnosťou je zmena dát cez API danej mikroslužby, v ktorej chceme meniť dáta. To pomôže dosiahnuť súdržnosť medzi mikroslužbami. Takýto cyklus prístupu do inej databázy je zobrazený na diagrame v obrázku 27.[15]

Obr. 27: Diagram prístupu do databázy inej mikroslužby

8.2.3 Nové spustenie pozastavených požiadaviek

Do tejto implementácie som pridal možnosť nového spustenia pozastavených požiadaviek. To mohlo byť spôsobené dlhým spracovaním, kedy mikroslužba nemala dostatok zdrojov pre jej vykonanie. Väčšinou býva toto pozastavenie kvôli dátam, kedy nie sme schopní uložiť údaje do pamäte, prípadne chceme urobiť zmeny v dátach, ale daná operácia zlyhá. V takýchto prípadoch môžeme skúsiť vykonať operáciu znova, pretože predpokladáme, že sa zdroje obnovia po určitom čase, prípadne môže load balancer poslať takúto požiadavku na inú mikroslužbu. Mali by sme ale

brať ohľad na počet nových spustení, pretože veľké množstvo opakovaní môže zabrániť obnoveniu webovej aplikácie. V distribuovaných systémoch môže opakované spustenie vyvolať kaskádový efekt, pretože môže spustiť niekoľko ďalších požiadaviek a opakovaní. Pre minimalizáciu týchto problémov by sme mali obmedziť počet opakovaní, prípadne robiť medzi nimi väčšie časové rozostupy až do nejakého limitného času. Ďalšou možnosťou je opakované spustenie zo strany klienta (webový prehliadač, iné mikroslužby), kedy klient nevie, či operácia zlyhala pred alebo po zaslaní novej požiadavky. V taktomto prípade sa môže stať, že daná požiadavka bude vykonaná aj druhý krát. Napríklad, ak sa opakuje platobná požiadavka, nemal by sa zákazník účtovať dva krát. Pri takýchto prípadoch sa každej požiadavke priradí špeciálny kľúč, ktorý zabezpečí vykonanie danej operácie len raz.[16]

8.3 Model distribuovanej aplikácie

V tejto časti si popíšeme jednotlivé časti namodelovanej siete distribuovanej aplikácie. Pri návrhu siete som vychádzal z diagramu 24. Tento diagram som ešte rozšíril o horizontálne škálovanie, čím vznikne viacej serverov, na ktorých beží takáto aplikácia. Pri popise je vždy uvedené len jedno miesto alebo prechod, bez priraďujúceho čísla. Skript tohto modelu je v prílohách.

8.3.1 Load balancer

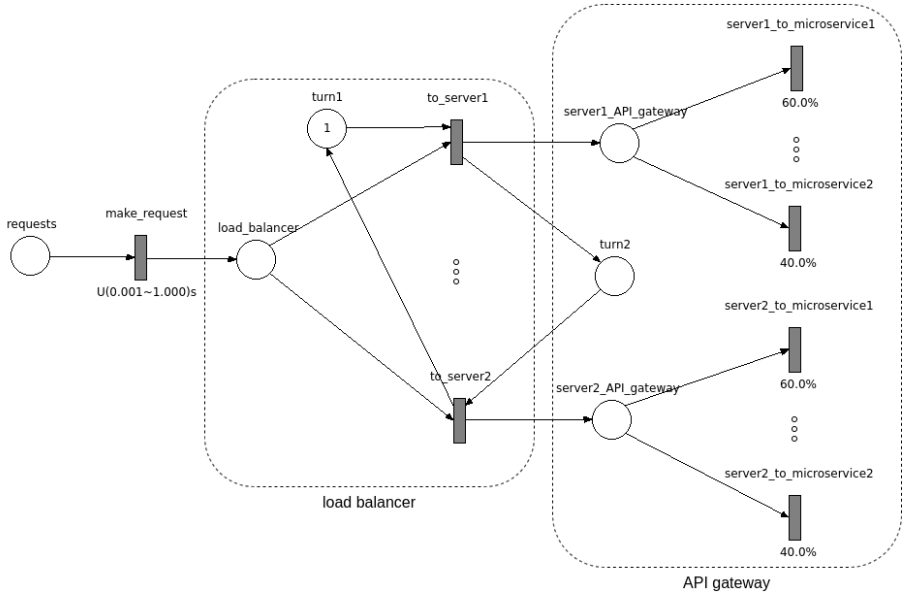
Pri návrhu load balanceru som využil metódu **round robin**, ktorá je popísaná v časti 8.1.1. Chovanie tejto metódy sa dá simulovať pomocou skupiny prechodov, v mojom prípadne prechodov *to_server*. Tieto prechody sa pomocou pomocných miest *turn* spúšťajú po rade, ako to je pri férovej metóde round robin. To zabezpečí, že každý takýto prechod zoberie zo vstupného miesta približne rovnaký počet tokenov, čo teda simuluje rovnomerné rozloženie požiadaviek od klientov medzi servermi. Táto sieť je zobrazená v ľavej časti obrázku 28.

Popis miest

- *requests* – Ide o vstupné miesto do celej aplikácie. Počet tokenov v tomto mieste vyjadruje počet požiadaviek od klientov. Z tohto miesta sa odoberajú tokeny do prechodu *make_request*.
- *load_balancer* – Ide o miesto, kde čakajú tokeny na priradenie k jednotlivým serverom. Z tohto miesta sa odoberajú tokeny do prechodov *to_server* podľa metódy round robin.
- *turn* – Ide o pomocné miesta, ktoré zabezpečia spúšťanie prechodov *to_server* po rade. Všetky tieto miesta obsahujú vždy len jeden token, ktorý prechádza medzi nimi.

Popis prechodov

- *make_request* – Tento prechod zabezpečuje simuláciu príchodu požiadaviek od používateľov. Ide o časovaný prechod, kde pri zmene jeho parametrov meníme aj frekvenciu prichádzajúcich požiadaviek. Tento prechod posiela tokeny do miesta *load_balancer*.
- *to_server* – Tieto prechody slúžia pre distribúciu požiadaviek z miesta *load_balancer* na jednotlivé servery. Jedným z jeho vstupov je aj miesto *turn*, ktoré zabezpečuje cyklické striedanie spúšťania týchto prechodov. Tento prechod pridáva tokeny do miesta *server_API_gateway*.



Obr. 28: Petriho sieť load balanceru

8.3.2 API brána

Pri návrhu API brány som vychádzal z podobnej siete, aká je aj pri load balanceri, ale časované prechody som nahradil prechodmi s pravdepodobnosťou. Pomocou zmeny tejto pravdepodobnosti sa mení počet požiadaviek, ktoré pôjdu do jednotlivých mikroslužieb. To zabezpečí, aby nebola každá mikroslužba zaťažovaná rovnako, čím sa dá simulovať reálnejšie správanie takéhoto systému. Podrobnejšie informácie o API bráne sú v časti 8.2.1. Petriho sieť takejto brány je v pravej časti obrázku 28.

Popis miest

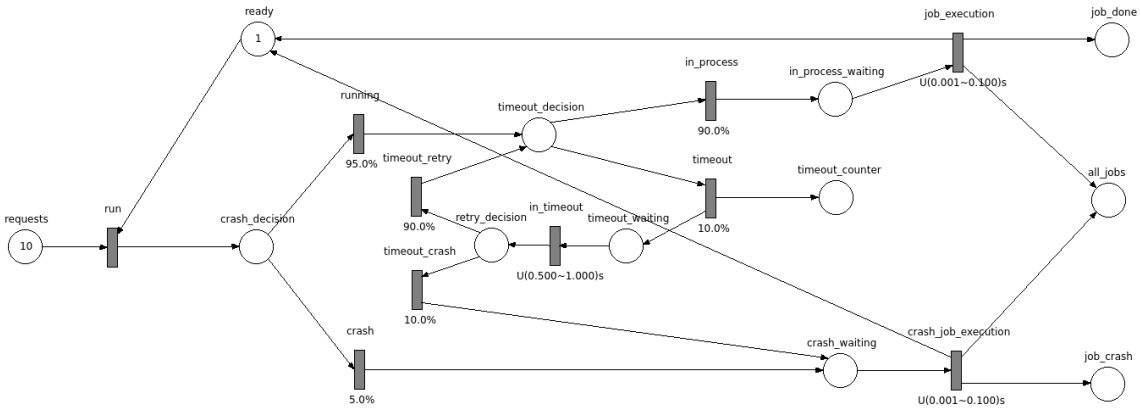
- *server_API_gateway* – Toto miesto slúži ako vstupné miesto pre jednotlivé mikroslužby. Každý server obsahuje práve jedno takéto miesto. Z tohto miesta sa odoberajú tokeny do prechodov *server_to_microservice*.

Popis prechodov

- *server_to_microservice* – Tieto prechody slúžia pre distribúciu požiadaviek z miesta *server_API_gateway* medzi jednotlivé mikroslužby. Ide o stochastické prechody, kde zmenou ich pravdepodobnosti môžeme simulovať rôzne zaťaženia jednotlivých mikroslužieb. Z tohto prechodu sa pridávajú tokeny do miesta *requests* jednotlivých mikroslužieb.

8.3.3 Mikroslužba

Najpodstatnejšou časťou návrhu distribuovanej webovej aplikácie je práve návrh mikroslužby. V tejto sieti sa nachádza veľa komponentov, ktoré sa snažia čo najpresnejšie vystihnúť správanie takejto aplikácie. Pre lepšie vystihnutie podstaty tokenov v miestach ich budem nazývať požiadavkami. Pri návrhu tejto siete som zvolil dve hlavné vetvy, a to vrchná vetva "running" a spodná vetva "crash". Požiadavky sa medzi ne delia hneď v prvej konfliktnej skupine. Ďalej som do vetvy "running" zaradil možnosť pozastavenia spracovania požiadavky, nazvime túto vetvu "timeout", čím je možné simulovať napríklad príliš dlhé spracovanie požiadavky. Tie následne môžu byť vyhodnotené ako znova nespustiteľné, takže pôjdu do vetvy "crash", alebo sa znova spustia a pôjdu znova do vetvy "running". Na konci každej vetvy som pridal časovaný prechod, ktorý simuluje dobu spracovania požiadavky. V tejto dobe je zaradená aj komunikácia s databázou. Petriho sieť mikroslužby je zobrazená na obrázku 29.



Obr. 29: Petriho sieť mikroslužby

Popis miest

- *requests* – Ide o vstupné miesto do Petriho siete mikroslužby. Toto miesto slúži ako fronta pre čakajúce požiadavky od klientov. Počet tokenov v tomto mieste ovplyvňujú najmä časované prechody *job_execution*, *in_timeout* a prechody v sieti load balanceru.

- *ready* – Toto miesto slúži pre overenie, či mikroslužba práve spracováva požiadavku z miesta *requests*. Toto miesto obsahuje najviac jeden token, pričom ak sa v mieste nachádza token, je umožnený prechod *run* a požiadavka sa môže spracovať. Ak sa v tomto mieste nenachádza token, mikroslužba je v stave spracovávania.
- *crash_decision* – Ide o vstupné miesto pre dvojicu prechodov *running* a *crash*, ktoré slúžia pre separáciu požiadaviek, ktoré z nejakého dôvodu nebude možné spracovať. Z tohto miesta ide token buď do vetvy running alebo vetvy crash.
- *timeout_decision* – Ide o vstupné miesto pre dvojicu prechodov *in_process* a *timeout*, ktoré slúžia pre separáciu požiadaviek, ktoré sa napríklad pre dlhé spracovanie môžu pozastaviť.
- *timeout_counter* – Toto miesto slúži pre sčítavanie požiadaviek, ktoré vstúpia do prechodu *timeout*.
- *retry_decision* – Ide o vstupné miesto pre dvojicu prechodov *timeout_retry* a *timeout_crash*, ktoré slúžia pre rozdelenie požiadaviek tak, že buď budú znova spustené, alebo budú brané ako nespustiteľné.
- *in_process_waiting* – V tomto mieste čakajú požiadavky vo vetve running na dokončenie ich spracovávania.
- *crash_waiting* – V tomto mieste čakajú požiadavky vo vetve crash na dokončenie ich spracovávania.
- *job_done* – V tomto mieste sa sčítavajú všetky požiadavky z vetvy running, ktoré sa spracovali bez komplikácií.
- *job_crash* – V tomto mieste sa sčítavajú všetky požiadavky z vetvy crash, v ktorých sa vyskytol nejaký problém.
- *all_jobs* – Toto miesto slúži pre sčítavanie všetkých požiadaviek, ktoré sú spracovávané danou mikroslužbou. Ide o výsledný súčet tokenov v miestach *job_done* a *job_crash*.

Popis prechodov

- *run* – Ide o prechod, ktorý umožňuje požiadavkám z miesta *requests*, aby mohli byť spracované. Tie následne posiela do miesta *crash_decision*.
- *running* – Ide o prechod, ktorý má pravdepodobnosť, s akou pôjdu požiadavky do vetvy running. Pomocou zmeny hodnoty pravdepodobnosti ovplyvňujeme počet požiadaviek, ktoré pôjdu cez túto vetvu. Tento prechod je v konfliktnej skupine s prechodom *crash*.

- *crash* – Ide o prechod, ktorý má pravdepodobnosť, s akou pôjdu požiadavky do vetvy crash. Tento prechod má rovnaké vlastnosti ako prechod *running*.
- *in_process* – Tento prechod má pravdepodobnosť, s akou pôjdu požiadavky do finálneho stavu spracovania vo vetve running. Tento prechod je v konfliktnej skupine s prechodom *timeout*.
- *timeout* – Tento prechod má pravdepodobnosť, s akou sa požiadavky dostanú do stavu, kedy budú z nejakého dôvodu pozastavené. Prechod má rovnaké vlastnosti ako prechod *in_process*
- *timeout_retry* – Ide o prechod, ktorý zabezpečuje opätovné spustenie požiadavky s nejakou pravdepodobnosťou, ak bola pozastavená. Tento prechod je v konfliktnej skupine s prechodom *timeout_crash*.
- *timeout_crash* – Tento prechod sa stará o to, aby sa pozastavené požiadavky dostali s nejakou pravdepodobnosťou do vetvy crash.
- *job_execution* – Ide o časovaný prechod, ktorým sa simuluje spracovanie požiadavky vo vetve running.
- *crash_job_execution* – Ide o časovaný prechod, ktorým sa simuluje spracovanie požiadavky vo vetve crash.

8.4 Simulácie a overenia pre rôzne scenáre

V tejto časti si ukážeme simulácie, pomocou ktorých overíme správanie vytvorenej siete. Pre väčšinu simulácií budeme používať parametre zo základného nastavenia popísaného nižšie, prípadné odlišnosti parametrov sú uvedené pri každej simulácií zvlášť.

8.4.1 Základné nastavenie parametrov miest a prechodov

Toto nastavenie budem používať pri väčšine simulácií. Tieto nastavenia sú rovnaké pre každú mikroslužbu, pokiaľ ich je viacero. Nejde o nastavenia pre konkrétnu aplikáciu, ale pre vystihuntie správania tohto modelu sú nastavené na nasledovné hodnoty.

Nastavenie parametrov miest

Ide o prvotné nastavenie miest, teda o počiatočný stav. Miesta sú nastavené podľa tabuľky 1.

Miesto	Hodnota
load_balancer	500

Tab. 1: Počiatočné nastavenie miest

Nastavenie parametrov časovaných prechodov

Časované prechody majú dva parametre, a to minimálny a maximálny čas spustenia. Pri simuláciách sa vždy berie hodnota rovnomerného rozdelenia z týchto parametrov. V tabuľke 2 sú uvedené tieto parametre pre každý časovaný prechod. Tieto parametre môžu byť odlišné pri viacerých serveroch a mikroslužbách.

Prechod	t_min [s]	t_max [s]
make_request	0,001	0,01
job_execution	0,005	0,05
crash_job_execution	0,005	0,05

Tab. 2: Nastavenie časovaných prechodov

Nastavenie parametrov prechodov s pravdepodobnosťou

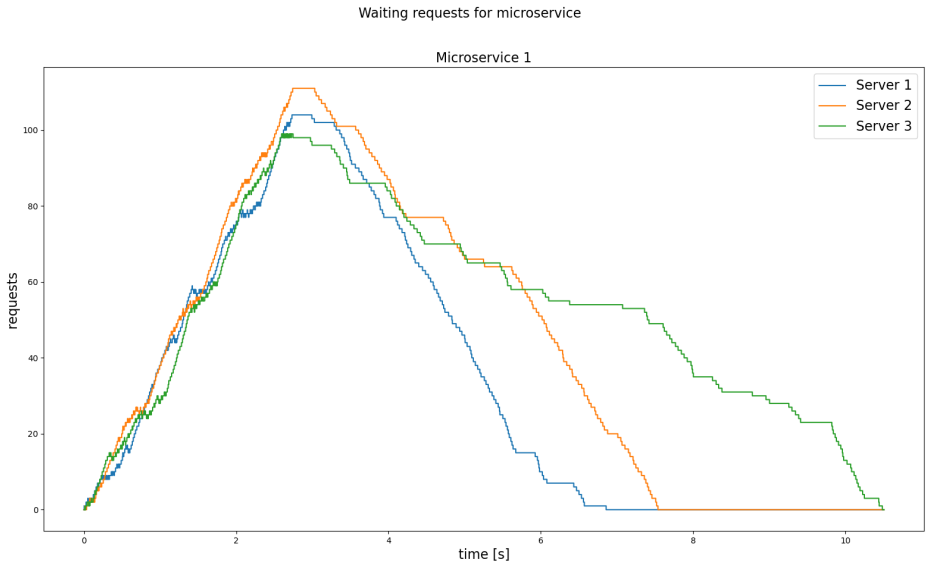
Ide o nastavenie prechodov s pravdepodobnosťou, podľa ktorej budú spustené. V tabuľke 11 sú uvedené pravdepodobnosti pre každý takýto prechod. Tieto pravdepodobnosti sa môžu meniť pri viacerých serveroch a mikroslužbách.

Prechod	Pravdepodobnosť [%]
to_microservice	50
running	95
crash	5
in_process	90
timeout	10
timeout_retry	80
timeout_crash	20

Tab. 3: Nastavenie prechodov s pravdepodobnosťou

8.4.2 Overenie rovnomerného rozdelenia požiadaviek pomocou load balancera

Pri tomto overení uvažujme len s jednou mikroslužbou a viacerými servermi, v tomto prípade to budú tri servery. Máme teda sieť s parametrami, ktoré majú hodnoty podľa základného nastavenia. Použitá metóda **round robin** zabezpečí, že každý server dostane rovnaký počet požiadaviek. Toto overenie je tiež pekne vidieť na grafe závilosti počtu čakajúcich požiadaviek mikroslužby na čase, ktorý je zobrazený na obrázku 30. Mikroslužba bola rovnako nastavená na všetkých serveroch. Začiatočná časť krivky, kde má stúpajúci charakter, reprezentuje prichádzanie požiadaviek na server. Tieto krivky sú podobné pri všetkých troch serveroch, kde menšie odchýlky sú spôsobené rovnomerným rozdelením parametrov pri časovaných prechodoch. Týmto môžeme potvrdiť správne správanie, a teda aj dizajn load balancera.



Obr. 30: Graf závilosti počtu čakajúcich požiadaviek mikroslužby na čase

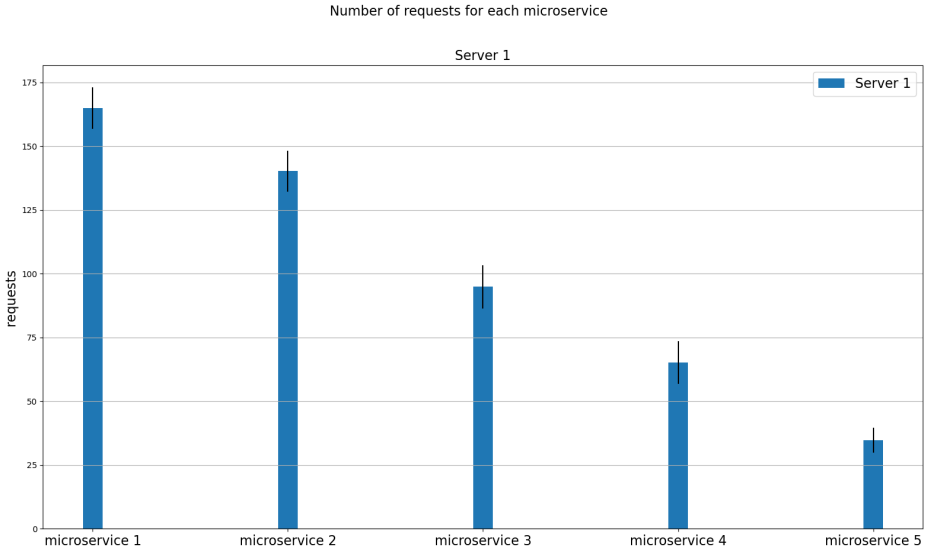
8.4.3 Overenie rozdelenia požiadaviek pre každú mikroslužbu

Rozdelenie požiadaviek v tomto modeli zabezpečuje nerovnomerné vyťaženie mikroslužieb. Počet požiadaviek pre jednotlivé mikroslužby závisí na prechode *to_microservice*. Pri simulácii máme päť mikroslužieb a len jeden server. Máme základné nastavenie pre každú mikroslužbu, ale s rôznymi parametrami prechodov *to_microservice*. Rozdelenie požiadaviek je vidieť na obrázku 31, na ktorom vidíme jednotlivé mikroslužby, ktorým sú pridelené požiadavky podľa tabuľky 4. Pri prvej mikroslužbe je táto pravdepodobnosť najväčšia, čo znamená, že jej je pridelených najviac požiadaviek zo všetkých, ktoré sú nastavené na hodnotu 500. Tento graf zobrazuje výsledky z desiatich behov simulácií. Hodnoty sa pri každej simulácii mierne líšia, čo vidíme na

smerodajnej odchýlke. Takéto správanie je spôsobené stochastickými vlastnosťami Petriho sietí.

Mikroslužba	Pravdepodobnosť [%]
1	33
2	27
3	20
4	13
5	7

Tab. 4: Pravdepodobnostné rozdelenie požiadaviek pre mikroslužby



Obr. 31: Počet požiadaviek pridelených jednotlivým mikroslužbám

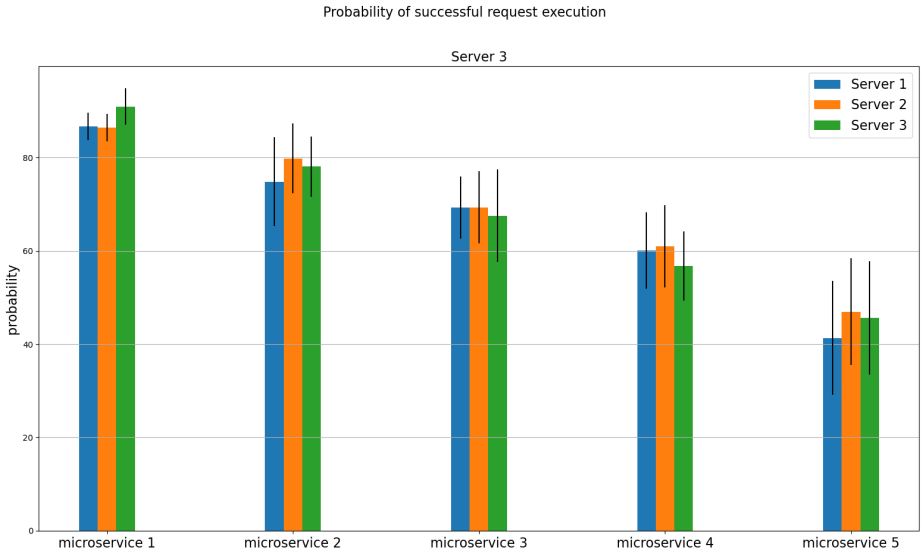
8.4.4 Overenie počtu úspešne spracovaných požiadaviek

Pri tomto overení sa pozrieme na to, koľko percent požiadaviek bude úspešne spracovaných. Sú to teda tie požiadavky, ktoré sú vo vetve "running". Pri tejto simulácii máme tri servery a päť mikroslužieb, ktoré majú rovnaké pravdepodobnosti pridelenia požiadaviek. To, či bude požiadavka vo vetve "running", zabezpečuje prechod s pravdepodobnosťou *running*. Pre každú mikroslužbu som túto pravdepodobnosť nastavil na rôzne hodnoty, ktoré sú uvedené v tabuľke 5. Musíme ale brať ohľad na to, že z vetvy "running" do vetvy "crash" môže požiadavka prejsť, ak sa dostane do pozastavenia. Na obrázku 32 môžeme vidieť štatistiky desiatich behov simulácie pre vetvu *running*. Môžeme si všimnúť, že ak je veľká pravdepodobnosť prechodu *running*, ako je to pri prvej mikroslužbe, hodnoty sa pri jednotlivých behoch veľmi nemenili, ako naznačuje aj smerodajná odchýlka. Čím je ale táto pravdepodobnosť

menšia, smerodajná odchýlka má vyššiu hodnotu. Toto správanie je z časti ovplyvnené vetvou "timeout", kedy sa pri novom pokuse spustenia tejto požiadavky môže dostať do vetvy "crash".

Mikroslužba	Pravdepodobnosť [%]
1	90
2	80
3	70
4	60
5	50

Tab. 5: Pravdepodobnosť rozdelenia požiadaviek do vetvy running



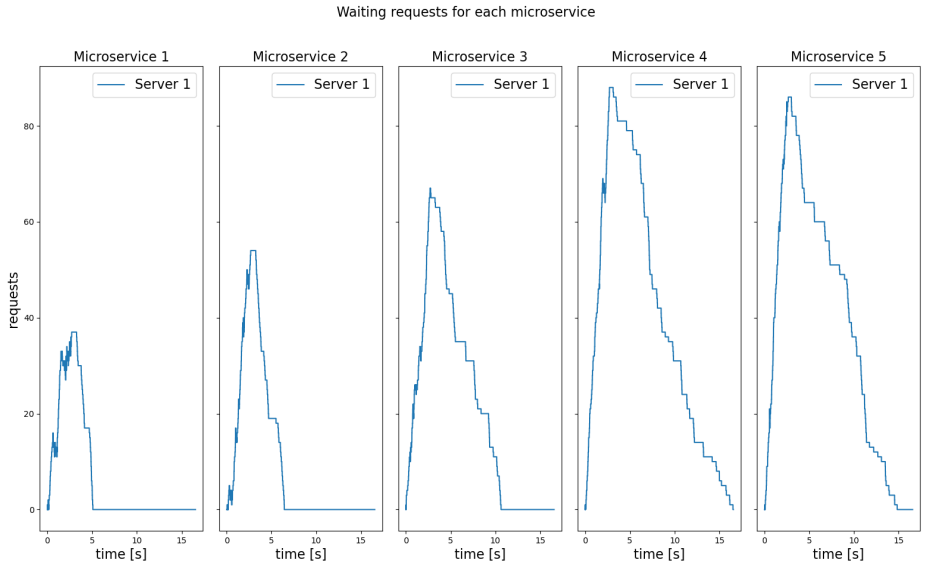
Obr. 32: Pravdepodobnosť správne vykonaných požiadaviek

8.4.5 Overenie počtu požiadaviek, ktoré boli pozastavené

Počet pozastavených požiadaviek ovplyvňuje prechod *timeout*. Tento prechod má pravdepodobnosť, s akou sú tieto požiadavky pozastavené. Pozastavená požiadavka je následne znova spustená, čo má za následok jej dlhšie spracovanie. Zmenou pravdepodobnosti sa teda zmení aj počet čakajúcich požiadaviek. To môžeme vidieť na obrázku 33, kde je na serveri päť mikroslužieb, ktoré majú nastavené parametre prechodov *timeout* podľa tabuľky 6. Všetky mikroslužby majú rovnakú pravdepodobnosť pridelenia požiadaviek. V grafe môžeme vidieť, že čím je väčšia pravdepodobnosť pozastavených požiadaviek, tým sa zvýši počet čakajúcich požiadaviek a predĺži sa čas ich spracovania.

Mikroslužba	Pravdepodobnosť [%]
1	10
2	20
3	30
4	40
5	50

Tab. 6: Pravdepodobnostné rozdelenie požiadaviek pre mikroslužby



Obr. 33: Pravdepodobnosť správne vykonaných požiadaviek

8.4.6 Preťaženie servera a mikroslužieb

Preťaženie servera môžeme simulovať pomocou zmeny parametrov prechodu *to_server*. Je to časovaný prechod, pri ktorom, ak zmeníme čas odpalénia prechodu na menšiu hodnotu ako je čas pre vykonanie požiadavky na mikroslužbe, dostaneme sa do situácie, kedy fronta čakajúcich požiadaviek bude prijímať viacej požiadaviek, ako je schopná spracovať. Simuláciu budeme vykonávať na dvoch serveroch, a oba parametre prechodov *to_server* sú nastavené podľa tabuľky 7. Máme tri mikroslužby, ktoré majú pravdepodobnosť pridelenia požiadaviek podľa tabuľky 8. Máme nastavené časy vykonania požiadaviek tiež podľa tejto tabuľky. Parametre pre príchod požiadaviek majú teda menšie časy ako časy vykonania požiadaviek, aby bola umožnená simulácia preťaženia. Na obrázku 34 môžeme vidieť viaceré situácie. Prvá mikroslužba, ktorá má najväčšiu pravdepodobnosť pridelenia požiadaviek, sa dostala do "preťaženia". To naznačuje veľký počet požiadaviek, ktorý čaká na spracovanie. Druhá mikroslužba má menšiu pravdepodobnosť pridelenia požiadaviek a teda počet čakajúcich požiadaviek sa výrazne znížil. Pri tejto mikroslužbe je len menšie "preťaženie" na druhom serveri. Posledná mikroslužba má najmenší počet pridelených

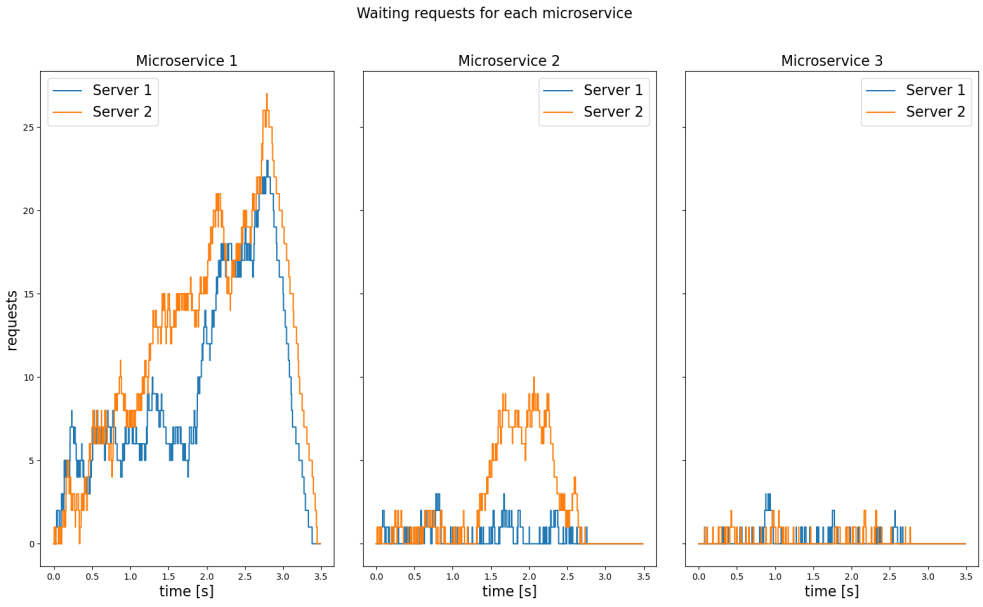
požiadaviek. Môžeme vidieť, že čakajúca požiadavka je väčšinou jedna, čo znamená, že mikroslužba ich stíha spracovať pred príchodom ďalšej. Ide teda o normálny beh bez "preťaženia".

Prechod	t_min [s]	t_max [s]
to_server	0,001	0,01
job_execution	0,005	0,05

Tab. 7: Nastavenie parametrov pre simuláciu preťaženia

Mikroslužba	Pravdepodobnosť [%]
1	50
2	30
3	20

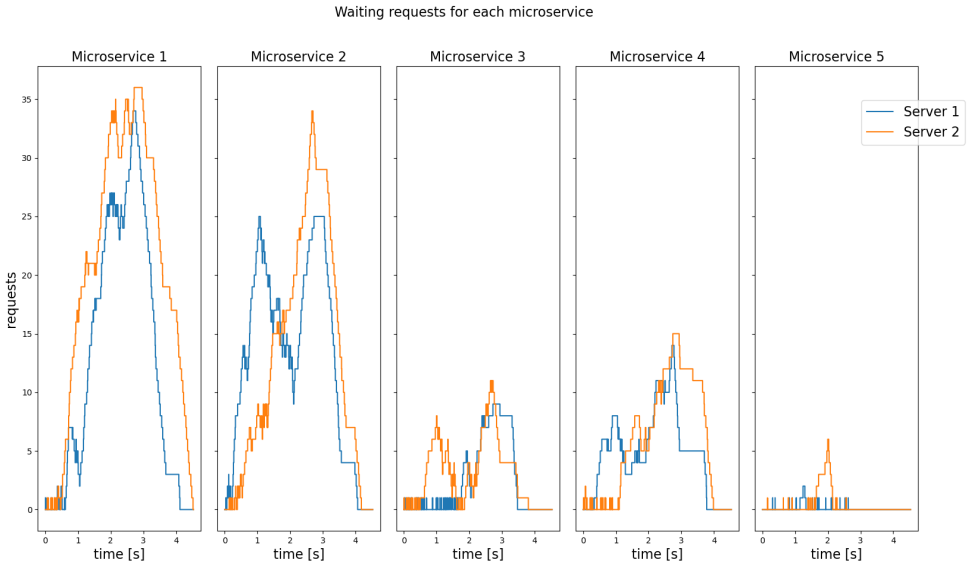
Tab. 8: Pravdepodobnostné rozdelenie požiadaviek pre mikroslužby



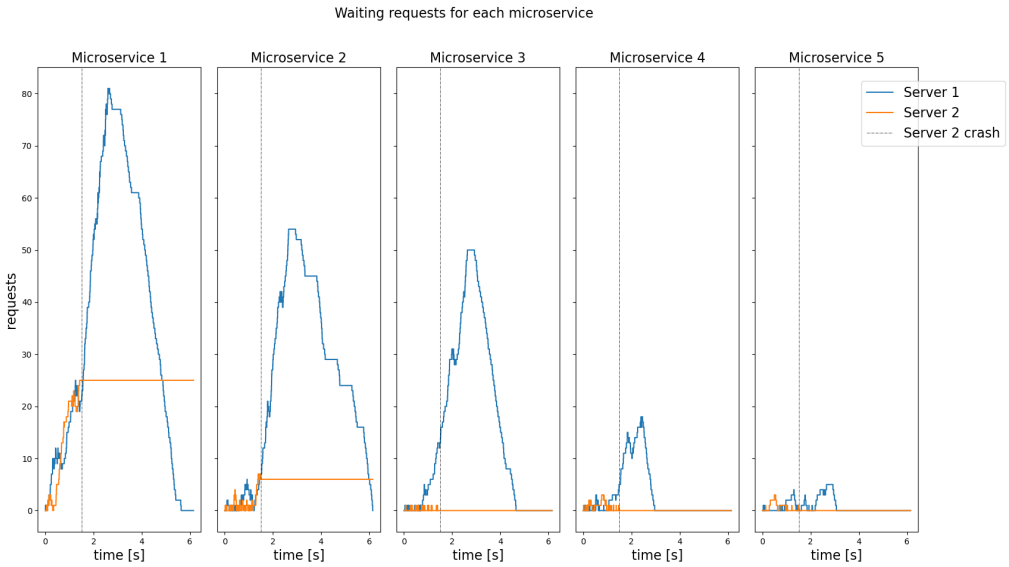
Obr. 34: Preťaženie servera

8.4.7 Zlyhanie servera a mikroslužieb

Uvažujme situáciu, pri ktorej máme dva servery a päť mikroslužieb. Každá mikroslužba má pridelené požiadavky podľa tabuľky 4. Pri simulácii zlyhania druhého servera, čiže aj všetkých jeho mikroslužieb, ktoré na ňom bežia, som musel pridať do petriho siete tohto modelu časovaný prechod s miestom, z ktorého idú hrany do prechodu *to_server2* a všetkých prechodov *run* pri každej mikroslužbe na tomto serveri. Na obrázku 35 môžeme vidieť normálny priebeh webovej aplikácie, kedy krivky počtu čakajúcich požiadaviek medzi oboma servermi majú podobný priebeh. Zlyhanie druhého servera môžeme vidieť na obrázku 36, kedy v čase 1,5 sekundy prestane prijímať požiadavky. Požiadavky ktoré ostali vo fronte čakajúchich požiadaviek sa nespracujú, čo je vidieť na horizontálnej čiare pri priebehu na druhom serveri. Od tohto času sa všetky požiadavky presmerujú len na prvý server, čo má za následok jeho preťaženie. To je pekne vidieť napríklad na druhej a štvrtej mikroslužbe, kedy do zlyhania obe krivky neprekročia istú hranicu čakajúcich požiadaviek, ale po zlyhaní sa táto hranica na prvom serveri zvýši. Pri porovnaní oboch obrázkov si môžeme všimnúť aj maximálne hodnoty čakajúcich požiadaviek, kde pri normálnom priebehu sú polovičné oproti behu so zlyhaním.



Obr. 35: Normálny priebeh webovej aplikácie bez zlyhania



Obr. 36: Zlyhanie druhého servera webovej aplikácie

9 Implementácia 2: Veľký sieťový výpočtový systém – platforma BOINC s projektom Folding@home

9.1 Ako funguje platforma BOINC

Projekt na platforme BOINC[1] zodpovedá nejakej organizácii alebo výskumnej skupine, ktorá vykonáva výpočty na verejných zdrojoch. Je teda primárne určený pre vedcov, nie pre programátorov a slúži na vedecké výpočty. Takýto projekt má jednu webovú stránku, ktorá je jeho hlavnou stránkou a tiež slúži ako adresár pre plánovacie servery. Projekt má relačnú databázu, v ktorej sú uložené popisy aplikácií, klientov, výsledkov, účtov, tímov atď. Projekt môže obsahovať viacero aplikácií, ktoré riešia daný problém. Účastníci projektu, teda klienti, si vyberajú projekt, na ktorom sa chcú podieľať. Projekt si vyberú prostredníctvom registrácie na jeho stránke a stiahnutím BOINC klienta.[17] Ďalšia možnosť je stiahnutie BOINC softvéru, ktorý obsahuje všetky aplikácie projektov. Fungovanie BOINC softvéru sa dá jednoducho vysvetliť pomocou nasledujúcich piatich bodov:[18]

1. Počítač klienta dostane na spracovanie podúlohy projektu, ktoré rozdeľuje plánovací server projektu. Tie sa delia podľa výkonu počítača, a vždy dostane klient len podúlohy, ktoré je schopný vypočítať. Projekt môže obsahovať viaceré aplikácie, a plánovací server môže poslať ktorúkoľvek z nich.

2. Počítač klienta následne stiahne danú aplikáciu a jej vstupné súbory z databázy projektu. Pri aktualizácii aplikácií sa automaticky stiahnú aktuálne súbory.

3. Počítač následne spustí aplikáciu, ktorá vytvorí výstupné súbory.

4. Počítač nahrá tieto súbory do databázy projektu.

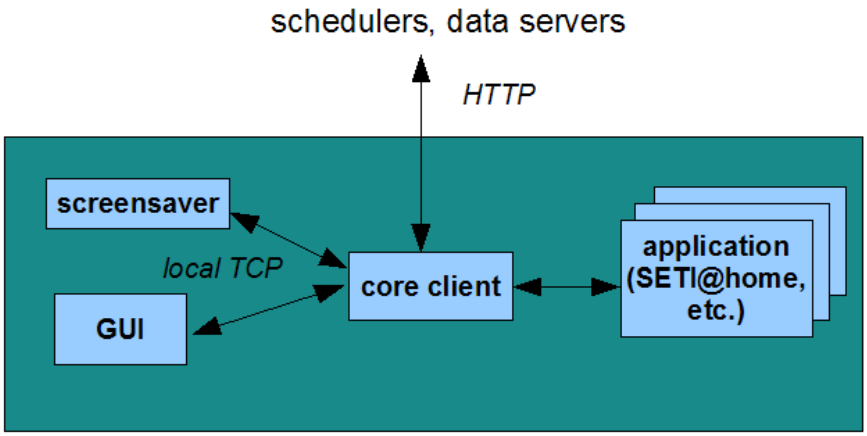
5. Na koniec, po niekoľkých hodinách alebo dňoch, v závislosti od výkonu počítača a nastavení, počítač ukončí výpočet všetkých úloh, ktoré mu dal plánovací server, a môže čakať na ďalšie úlohy.

9.1.1 Zloženie BOINC softvéru

BOINC softvér sa správa ako jeden program, ale skladá sa z viacerých programov. Plánovač a databáza sú teda súčasťou BOINC projektu, a sú inštalované a spravované jednotlivými projektami. Programy, ktoré sa vykonávajú na počítači klienta, sú zobrazené v zelenom rámiku na obrázku 37, a sú to:[18]

[1] Berkeley Open Infrastructure for Network Computing

- *core client* – Ide o program, obvykle označovaný len ako *klient*, ktorý sa stará o komunikáciu s projektovými servermi pre získavanie a nahrávanie dát. Tento program sa stará o spúšťanie a riadenie aplikácií.
- *applications* – Sú to aplikácie, ktoré vykonávajú samotné vedecké výpočty, teda výpočty na projekte. Ak má počítač viacero procesorov, je môžné spustenie viacerých aplikácií.
- *GUI* – Stará sa o grafické rozhranie, ktoré umožňuje ovládať a nastavovať *klienta*, napríklad môže pozastaviť beh aplikácií a podobne.
- *screensaver* – Ide o šetrič obrazovky, ktorý je spustený, ak sa nevykonáva žiadna iná činnosť na počítači okrem behu BOINC klienta. Tento šetrič je generovaný daným projektom, a nie každý projekt poskytuje takýto šetrič.



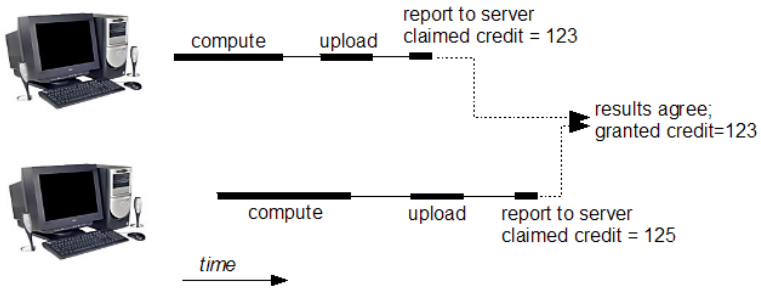
Obr. 37: Programy BOINC klienta[18]

9.1.2 Kredit

Server projektu sa stará tiež o to, koľko výpočtov bolo vykonaných na jednotlivých klientoch. Táto práca je zhrnutá do čísla, ktoré sa nazýva kredit. Pridelenie kreditu sa vykonáva podľa nasledujúcich krokov:[18]

1. Každá časť výpočtu, teda každá podúloha, je poslaná viacerým klientom.
2. Klient po nahratí výsledku na server dostane kredit, ktorého hodnota je vypočítaná na základe toho, koľko času spotreboval procesor klienta pre výpočet podúlohy.
3. Po nahratí aspoň dvoch výsledkov ich server porovná. Ak sa budú výsledky zhodovať, klienti dostanú kredit s menšou hodnotou, ktoré šli do porovnania.

Tieto kroky sú zobrazené na obrázku 38, kde po nahratí oboch výsledkov dostane každý z klientov kredit s hodnotou 123, čo je menšia z týchto dvoch kreditov. Kredit sa používa pre zostavenie rebríčka, ktorý zobrazuje, ako sa každý klient podieľal na danom projekte. Má teda za úlohu motivovať klientov pre použitie platformy BOINC, pretože ich zaujíma umiestnenie v tomto rebríčku. To napomáha zvýšeniu klientov, čo zabezpečí rýchlejšie spracovanie podúloh a dosiahnutie výsledkov.[17]



Obr. 38: Kredit klienta[18]

9.2 Zlyhania a replikácia výsledkov

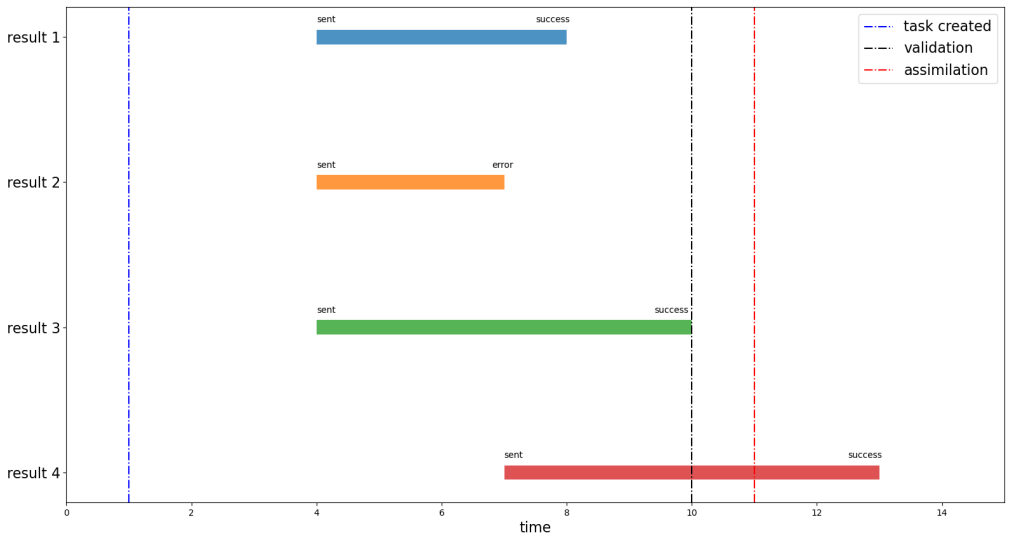
Typycky sa jednotlivé podúlohy posielajú viacerým klientom pre potreby verifikácie výsledku. Klienti tieto podúlohy spracujú, teda vykonajú ich výpočet a ten následne pošlú späť na BOINC server. Avšak je viacero možností, čo sa môže s takýmto výsledkom stať, a to:[19]

- klient vypočíta danú podúlohu správne a pošle ju naspať na server
- klient vypočíta danú úlohu nesprávne a pošle ju na server
- klient zlyhá pri sťahovaní alebo nahrávaní dát na server
- zlyhá aplikácia na strane klienta
- klient nikdy nepošle vypočítanú správu na server, pretože zlyhá alebo zastaví výpočet
- BOINC plánovač nie je schopný poslať danú podúlohu klientovi, pretože ten nemá dostatočné zdroje

Platforma BOINC teda vykonáva redundantný výpočet, čo znamená, že jednotlivé podúlohy vypočítajú viacerí klienti. Výsledky týchto podúloh sa porovnajú, a sú považované za správne len v prípade, ak sa budú v týchto výsledkoch zhodovať viacerí klienti. V niektorých prípadoch je nutné, aby bol vytvorený nový výsledok na novom klientovi. Porovnanie výsledkov je rozdelené do dvoch fáz a to:[19]

- **validácia** – Má dva možné stavy. Prvý stav je ten, že ak na server príde dostatočný počet výsledkov danej podúlohy, tak sa porovnajú, a skúsi sa nájsť súhlasné riešenie. Metódy, ktoré porovnávajú výsledky, by mali brať ohľad na menšiu variabilitu výsledkov vzhľadom na rôzne platformy klientov, a metódy určovania súhlasného riešenia, sú dané aplikáciou projektu. Druhý stav je taký, že ak výsledok príde po nájdení súhlasného riešenia, tento výsledok sa s ním porovná, a v prípade zhody dostane klient kredit.
- **asimilácia** – Ide o mechanizmus, pomocou ktorého je projekt informovaný o výsledku podúlohy. Pre každú podúlohu je vykonaný len raz. V prípade úspešného riešenia podúlohy, teda dosiahnutia súhlasného riešenia, sa takýto výsledok zapíše do databázy projektu. Ak riešenie podúlohy nebolo úspešné, projektová funkcia zapíše informácie o takomto riešení do logovacích záznamov, prípadne vykoná iné akcie.

Na obrázku 39 môžeme vidieť tieto dve fázy porovnania výsledkov. V čase 4 sa vytvoria traja klienti, ktorí sa pokúšajú o výpočet podúlohy. Klient *result 2* zlyhá v čase 7, a kvôli tomuto zlyhaniu sa vytvoril nový výsledok *result 4* na novom klientovi. Pre jednoduchosť predpokladajme, že pre úspešné porovnanie a nájdenie súhlasného výsledku sú nutné aspoň dve rovnaké riešenia. Validácia teda prebehne v čase 10, kedy sa vytvoril ďalší výsledok na klientovi *result 3*. Asimilácia prebehne po predchádzajúcom úkone.



Obr. 39: Porovnanie výsledkov podúlohy

9.2.1 Deadline pre výpočet podúlohy

Každá podúloha má zadaný parameter konečného času *delay_bound*, dokedy musí byť vypočítaná. Pri poslaní podúlohy J klientovi v čase t musí byť úloha vypočítaná do času t + *delay_bound(J)*. Ak daný klient do tohto času nesplní podúlohu J, pošle sa ďalšiemu klientovi. BOINC plánovač sa snaží posielať podúlohy len klientom, ktorí majú veľkú pravdepodobnosť jej dokončenia, a to na základe odhadu ich času dokončenia podúlohy a fronty úloh. Projekty môžu mať podúlohy, ktoré musia byť splnené skôr ako ostatné, napríklad z dôvodu podvýsledkov pre ďalší výpočet. Pre tieto úlohy môže byť parameter konečného času malý, čo obmedzí počet klientov, ktorým môže byť podproblém poslaný. Nie vždy je tento prístup správny, a môže spraviť opačný efekt toho, čo sme chceli dosiahnuť. Podúlohy, ktoré nemajú takú prioritu, môžu mať tento parameter veľký, napríklad niekoľko týždňov. Toto umožní aj "pomalším" klientom dokončiť výpočet, ale vstupné a výstupné údaje budú zaberať miesto v databázi dlhší čas.[17]

9.3 Projekt Folding@home

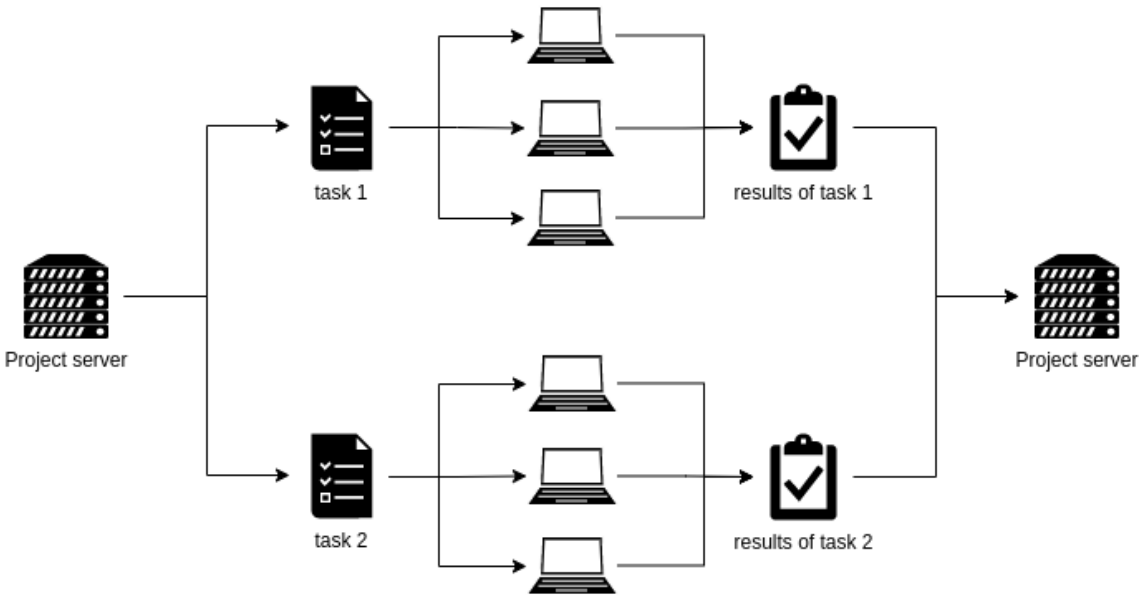
Folding@home je projekt na platforme BOINC, pomocou ktorého sa simuluje skladanie proteínov a ich pohyb, ktoré sa vyskytujú pri rôznych chorobách. Pomocou platformy BOINC sa poskytujú výpočtové zdroje pre takúto simuláciu od každého, kto sa do tohto projektu zapojí. Poznatky z týchto simulácií pomáhajú vedcom, aby lepšie porozumeli biologickým procesom, a tiež poskytujú nové informácie pre vývoj liečiv. Ide o jeden zo starších projektov, ale patrí medzi najúspešnejšie a najrýchlejšie. So zvýšeným záujmom o projekt kvôli pandémii COVID-19, dosiahol tento systém v apríli 2020 rýchlosť jeden exaFLOPS[2], čo je 10^{18} operácií za sekundu, a stal sa tak najrýchlejším výpočtovým systémom na svete. Takýto výpočtový výkon umožnil vedcom spustenie simulácií skladania proteínov na dlhší čas, ako to bolo doteraz možné.[20] Do tohto projektu sa zapojila aj univerzita VUT, a medzi všetkými českými univerzitami získala najviac bodov.[21]

9.4 Model projektu na platforme BOINC

V tejto časti si popíšeme jednotlivé časti namodelovanej siete projektu na platforme BOINC. Pri návrhu siete som vychádzal z diagramu 40. Na tomto diagrame sú vidieť jednotlivé fázy tvorenia a výpočtu podúloh. Pre lepšiu predstavivosť budem token nazývať podúlohou, teda nejakým problémom, ktorý sa snažíme vypočítať pomocou

[2] FLoating-point Operations Per Second

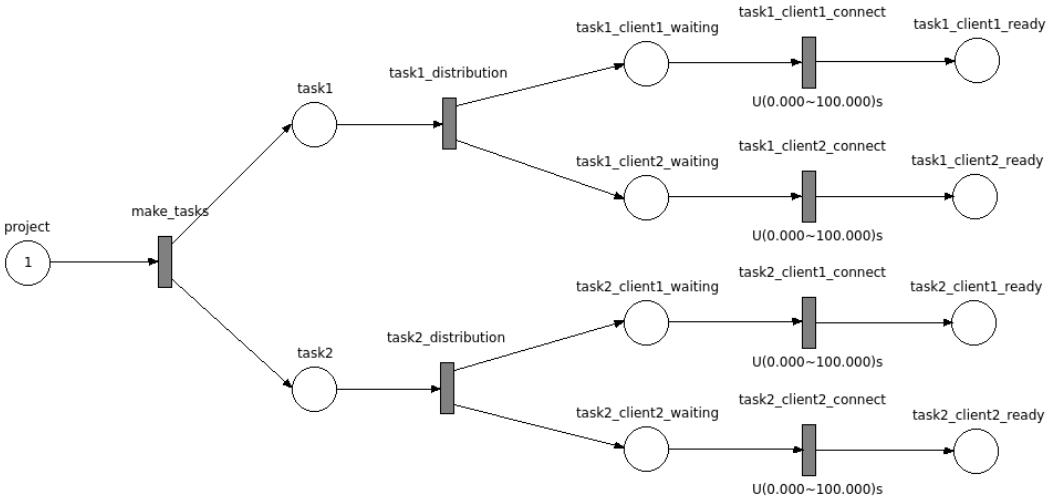
platformy BOINC. Pri dizajnovaní som bral ohľad na možné chyby, ktoré môžu nastať. Skript tohto modelu je v prílohách.



Obr. 40: Model projektu na platforme BOINC

9.4.1 Rozdelenie problému na časti a ich distribúcia

Ide o jednoduchú sieť, ktorá zabezpečí vytvorenie podúloh. Tieto podúlohy následne rozpošle viacerým klientom. Takáto sieť je zobrazená na obrázku 41. Všetky miesta aj prechody sú pri popise uvádzané bez predpony a pre každú úlohu a klienta len raz.



Obr. 41: Petriho sieť výroby úloh a ich distribúcia

Popis miest

- *project* – Ide o vstupné miesto do celej siete. Toto miesto reprezentuje projekt, teda problém, ktorý chceme vypočítať.
- *task* – Toto miesto reprezentuje časť problému z projektu, ktorá bude následne spracovaná viacerými klientami. Počet týchto miest teda znamená počet podúloh, na ktoré je problém rozdelený.
- *waiting* – V tomto mieste sa čaká pre pripojenie klienta.
- *ready* – Toto miesto je totožné so začiatočným miestom Petriho siete klienta.

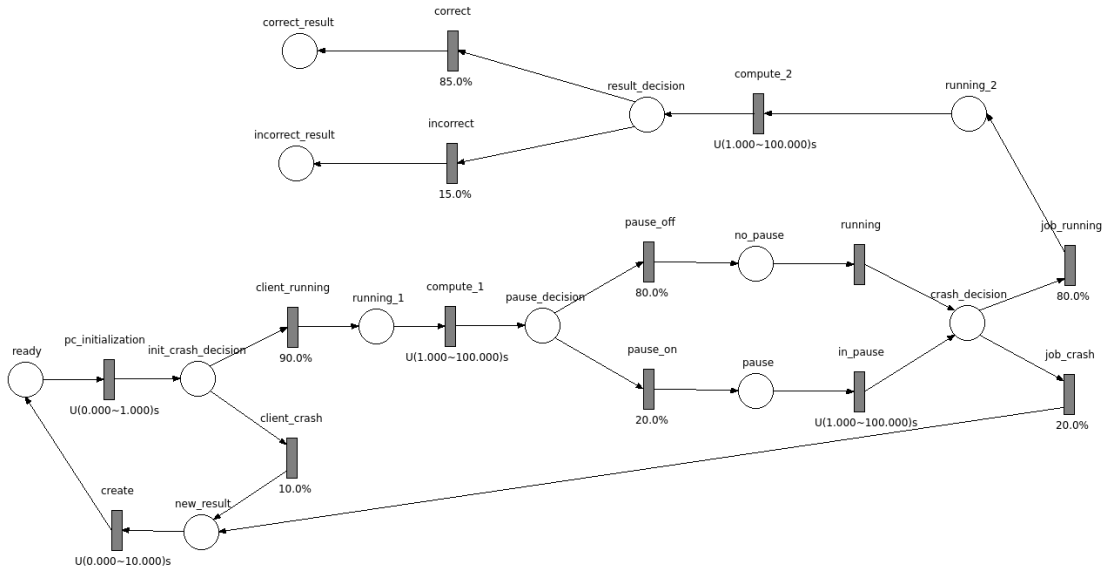
Popis prechodov

- *make_tasks* – Ide o prechod, ktorý vytvára podúlohy pre klientov. Berie token zo začiatočného miesta *project* a pridá ho do každého miesta *task*.
- *task_distribution* – Tento prechod slúži k rozposlaniu jednotlivých podúloh klientom. Týchto prechodov je rovnaký počet ako miest *task*.
- *connect* – Ide o časovaný prechod, ktorý slúži pre simuláciu rôznych časov pripojenia klientov. Nepredpokladá sa, že všetci klienti sú dostupní v rovnaký čas.

9.4.2 Klient

Ide o hlavnú časť siete v tomto modeli. Petriho sieť klienta je zobrazená na obrázku 42. Obsahuje celú simuláciu výpočtu klienta, v ktorej sú implementované nasledujúce body:

1. **inicializácia** – Zahŕňa prenos údajov ku klientovi, inicializáciu aplikácie a podobne, pri ktorej môže nastať zlyhanie.
2. **prvé spustenie** – Ide o prvú časť bezproblémového výpočtu, pri ktorej sa nepredpokladajú žiadne výpadky a pauzy.
3. **pauza klienta** – Simuluje možnosť pauzy klienta, ktorá nastane s nejakou pravdepodobnosťou. Po tomto kroku je istá pravdepodobnosť, že napríklad klient už nebude pokračovať vo výpočte alebo dôjde k nejakej chybe.
4. **druhé spustenie** – Ide o bezproblémový výpočet ako pri prvom spustení.
5. **rozhodnutie o výsledku výpočtu** – Konečný výsledok bude s nejakou pravdepodobnosťou správny.



Obr. 42: Petriho sieť výroby úloh a ich distribúcia

Popis miest

- *ready* – Ide o vstupné miesto do petriho siete klienta. Do tohto miesta vstupujú tokeny z prechodu *connect*, ktoré reprezentujú jednotlivé podúlohy projektu.
- *init_crash_decision* – V tomto mieste čakajú podúlohy na rozhodnutie, či budú spustené, alebo počas inicializácie klienta došlo k nejakej chybe.
- *running_1* – V tomto mieste čakajú úlohy na "prvý" výpočet podúlohy.
- *pause_decision* – Ide o miesto, v ktorom čakajú podúlohy na rozhodnutie, či budú alebo nebudú pozastavené klientom.
- *no_pause* – Token v tomto mieste znamená, že podúloha nebola pozastavená a pokračuje jej výpočet.
- *pause* – Token v tomto mieste znamená, že podúloha bola pozastavená, a čaká na vykonanie prechodu *in_pause*.
- *crash_decision* – Ide o miesto, v ktorom sa rozhoduje, či bude výpočet úlohy zastavený, alebo sa bude pokračovať v ďalšom výpočte.
- *new_result* – Do tohoto miesta idú tokeny z prechodov "crash", kde sa čaká na pripojenie nového klienta pre začatie nového výpočtu.
- *running_2* – Ide o miesto, kde sa čaká na ďalší výpočet podúlohy.
- *result_decision* – V tomto mieste sa rozhoduje, či vypočítaná podúloha bude mať "správny" výsledok.

- *correct_result* – Ide jedno z koncových miest Petriho siete klienta. Token v tomto mieste reprezentuje správne vypočítanú podúlohu.
- *incorrect_result* – Ide jedno z koncových miest Petriho siete klienta. Token v tomto mieste reprezentuje nesprávne vypočítanú podúlohu.

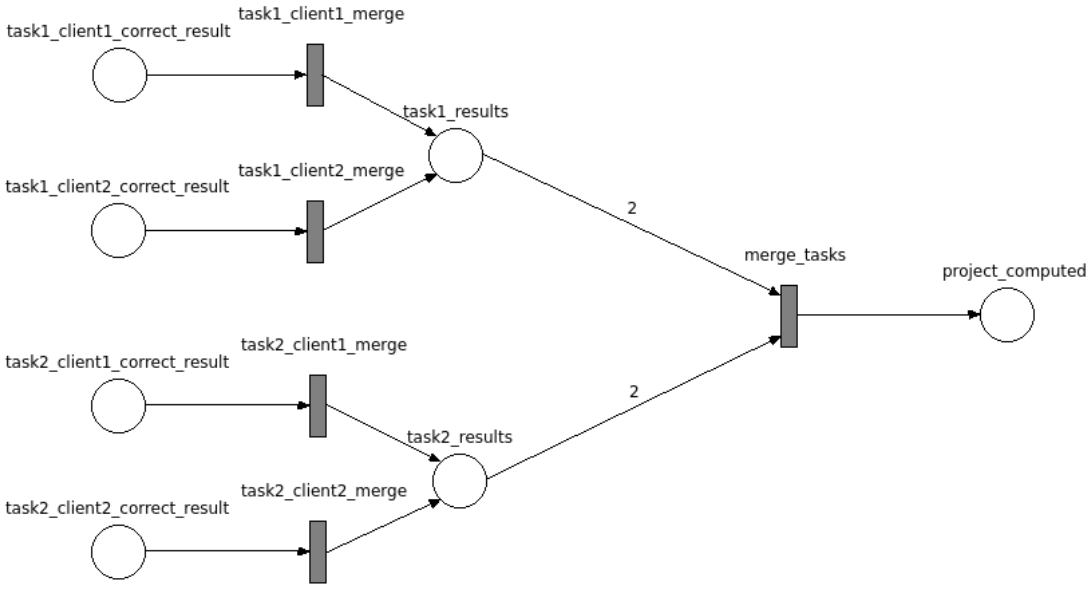
Popis prechodov

- *pc_initialization* – Ide o časovaný prechod, ktorý simuluje inicializáciu klienta, prenos podúlohy ku klientovi a podobne.
- *client_running* – Tento prechod je v konfliktnej skupine s prechodom *client_crash*. Má pravdepodobnosť, s akou pôjdu podúlohy do ďalšieho kroku výpočtu.
- *client_crash* – Tak ako pri predchádzajúcom prechode, aj tento má pravdepodobnosť, s akou bude znemožnený výpočet podúlohy. Toto znemožnenie môže byť spôsobené napríklad chybou pri prenose podúlohy ku klientovi, alebo pri inicializácii klienta.
- *compute_1* – Jedná sa o časovaný prechod, ktorý simuluje "prvý" výpočet podúlohy.
- *pause_off* – Tento prechod je v konfliktnej skupine s prechodom *pause_on*. Ide o prechod s pravdepodobnosťou, ktorá stanovuje, ako často pôjde podúloha do stavu, kedy je pozastavená.
- *pasue_on* – Ako pri predchádzajúcom prechode, aj tento má pravdepodobnosť, s akou sa dostane podúloha do stavu, kedy je pozastavená klientom.
- *running* – Tento prechod len posunie podúlohu do ďalšieho rozhodovania.
- *in_pasue* – Ide o prechod, ktorý simuluje dobu, v ktorej je výpočet podúlohy pozastavený.
- *job_running* – Ide o prechod, ktorý je v konfliktnej skupine s prechodom *job_crash*. Tento prechod má pravdepodobnosť, s akou bude výpočet podúlohy pokračovať do ďalšieho kroku výpočtu.
- *job_crash* – Podobne ako aj pri predchádzajúcom prechode, tento prechod má pravdepodobnosť, s aku bude znemožnený výpočet úlohy, napríklad pre zrušenie používateľom a podobne.
- *create* – Tento prechod zabezpečuje vytvorenie nového klienta pri zrušení výpočtu. Ide o časovaný prechod, ktorý simuluje čakanie na takéhoto klienta.

- *compute_2* – Ide o časovaný prechod, ktorý simuluje "druhý" výpočet podúlohy.
- *correct* – Tento prechod je v konfliktnej skupine s prechodom *incorrect*. Má pravdepodobnosť, s akou bude vypočítaná podúloha "správna". To zabezpečuje, že pôjdu len správne úlohy do Petriho sieťe porovnania, kde sa zbierajú len "správne" výsledky.
- *incorrect* – Podobne ako pri predchádzajúcom prechode, aj tento má pravdepodobnosť, s akou bude vypočítaná podúloha "nesprávna". Takáto podúloha už nejde ďalej do Petriho siete porovnania.

9.4.3 Porovnanie výsledkov medzi klientami a ich kompozícia

V tejto Petriho sieti som implementoval porovnanie vypočítaných podúloh od viacerých klientov. Na záver, ak dané podúlohy tvoria nejaký celok, na ktorý bol projekt rozložený, sú spojené, a môžeme predpokladať úspešný koniec výpočtu daného problému. Táto sieť je zobrazená na obrázku 43. Všetky miesta aj prechody sú pri popise uvádzané bez predpony a pre každú úlohu a klienta len raz.



Obr. 43: Petriho sieť porovnania úloh a ich zloženie

Popis miest

- *correct_result* – Ide o vstupné miesto do Petriho siete porovnania. Toto miesto je totožné s miestom *correct_result* v Petriho sieti klienta.

- *results* – V tomto mieste sa zhromažďujú výsledky jednotlivých podúloh od klientov. Pokiaľ je v tomto mieste určitý počet výsledkov, môžu byť porovnané, a môžu ísť do finálnej kompozície všetkých podúloh.
- *project_computed* – Ide o koncové miesto celej siete. Token v tomto mieste znamená úspešne vyriešený problém daného projektu.

Popis prechodov

- *merge* – Tento prechod zabezpečuje presun vypočítaných podúloh do miesta *results*, kde sa zhromažďujú výsledky danej podúlohy.
- *merge_tasks* – Ide o prechod, ktorý zabezpečuje kompozíciu jednotlivých podúloh, čím vznikne vypočítaný problém daného projektu. Tento prechod je spustený len v prípade, ak je v jednotlivých miestach *results* dostatočný počet výsledkov pre porovnanie.

9.5 Simulácie pre rôzne scenáre

V tejto časti si ukážeme simulácie, pomocou ktorých overíme správanie vytvorenej siete. Najskôr sú ale uvedené základné nastavenia miest a prechodov, podľa ktorých boli vykonané nasledovné simulácie.

9.5.1 Základné nastavenie parametrov miest a prechodov

Toto nastavenie bolo použité pri väčšine simulácií. Tieto nastavenia sú rovnaké pre každého klienta, pokiaľ ich je viacero.

Nastavenie parametrov miest
Ide o prvotné nastavenie miest, teda o počiatočný stav. Miesta sú nastavené podľa tabuľky 9.

Miesto	Hodnota
project	1

Tab. 9: Počiatočné nastavenie miest

Nastavenie parametrov časovaných prechodov
Časované prechody majú dva parametre, a to minimálny a maximálny čas spustenia. Pri simuláciách sa vždy berie hodnota rovnomerného rozdelenia z týchto parametrov. V tabuľke 10 sú uvedené tieto parametre pre každý časovaný prechod. Tieto parametre môžu byť odlišné pri viacerých klientoch podľa konkrétnej simulácie.

Prechod	t_min [s]	t_max [s]
connect	0	100
pc_initialization	0	1
compute_1	100	1000
in_pause	20	200
compute_2	200	1000
create	1	50

Tab. 10: Nastavenie časovaných prechodov

Nastavenie parametrov prechodov s pravdepodobnosťou

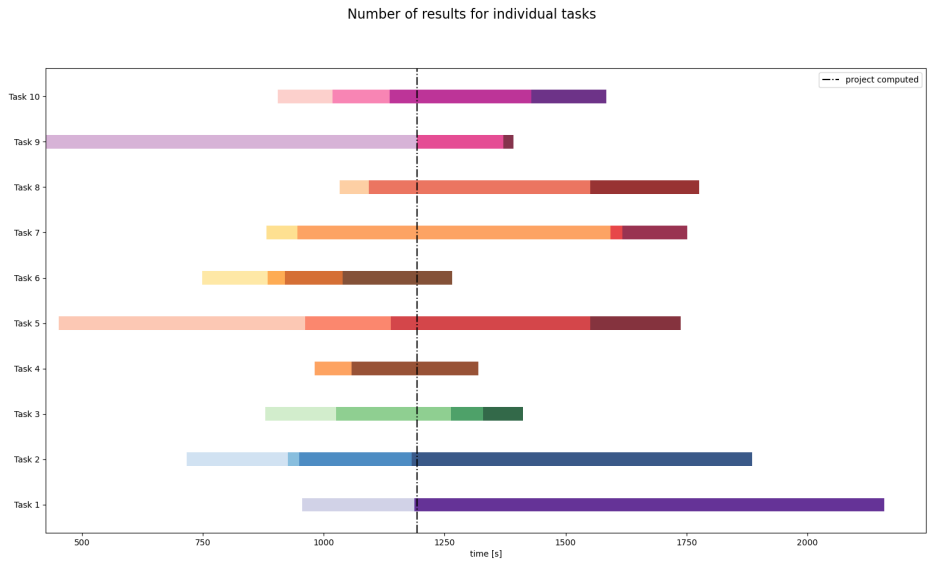
Ide o nastavenie prechodov s pravdepodobnosťou, podľa ktorej budú spustené. V tabuľke 11 sú uvedené pravdepodobnosti pre každý takýto prechod. Tieto parametre môžu byť odlišné pri viacerých klientoch podľa konkrétnej simulácie.

Prechod	Pravdepodobnosť [%]
client_running	90
client_crash	10
pause_off	80
pause_on	20
job_running	80
job_crash	20
correct	85
incorrect	15

Tab. 11: Nastavenie prechodov s pravdepodobnosťou

9.5.2 Simulácia pre základné nastavenie

V tejto simulácii ide o ukážku celého výpočtu problému podľa základného nastavenia miest a prechodov. Pri simulácii bolo použitých päť klientov, pričom problém bol rozdelený na desať podúloh. V grafe na obrázku 44 je časová závislosť počtu výsledkov od každého klienta pre každú podúlohu. Zmena počtu výsledkov je v grafe zobrazená ako farebné rozlíšenie pri jednotlivých stĺpcoch, kedy tmavšia farba znamená zvýšenie počtu výsledkov o jeden. Môžeme si všimnúť, že pri niektorých podúlohách je menší počet výsledkov, čo je spôsobené zobrazovaním len správnych výsledkov, ktoré vstupujú do porovnania. Ďalej si môžeme všimnúť, že výpočet úloh začína v rôznych časoch, a aj rôznosť času výpočtu, čo je vidieť na veľkosti daného farebného poľa.



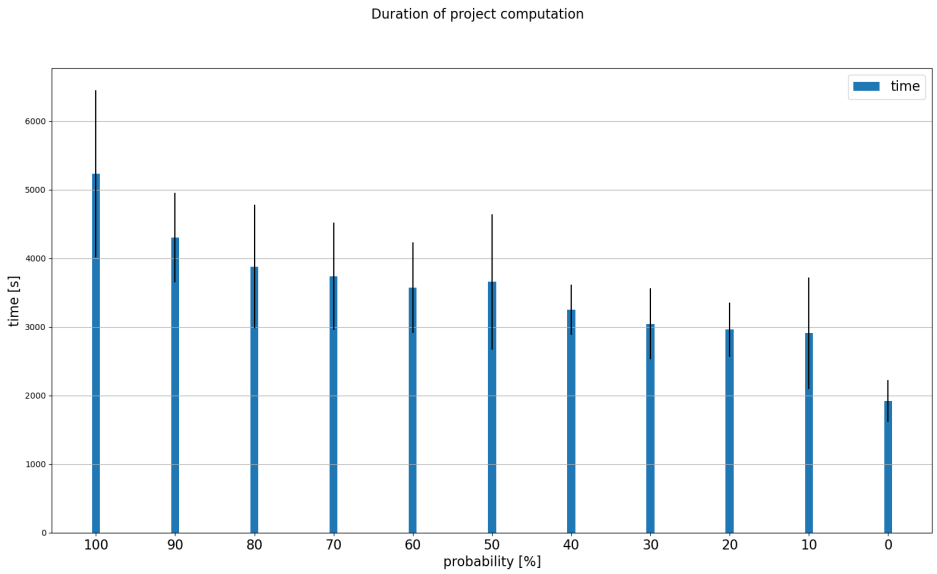
Obr. 44: Výpočet problému pri základnom nastavení

9.5.3 Pauza výpočtu užívateľom

Pauzu výpočtu používateľom ovplyvňuje dvojica prechodov *pause_off* a *pause_on*. Pri tejto simulácii bolo použitých päť klientov, pričom problém bol rozdelený na dve podúlohy. Na obrázku 45 sú zobrazené celkové časy výpočtu problému pre rôzne pravdepodobnosti spustenia prechodu *pause_on*, pričom vždy bol každý klient nastavený na túto pravdepodobnosť. Pre každú pravdepodobnosť bolo spravených desať behov simulácie, a v grafe je zobrazená vždy stredná hodnota z týchto behov a smerodajná odchýlka. Táto odchýlka má veľké rozpätie pre veľký rozostup parametrov časovaných prechodov a rovnomerné rozdelenie. V grafe je vidieť závislosť, že čím je pravdepodobnosť nižšia, klesá aj celkový čas výpočtu. Najviac je táto závislosť vidieť na okrajových hodnotách pravdepodobnosti, kde pri stredných hodnotách je táto závislosť menej výrazná pre stochastické vlastnosti Petriho sietí. Pre zvýraznenie týchto závislostí som zvýšil hodnotu časovaného prechodu *in_pause* podľa tabuľky 12, aby bola táto hodnota vyššia ako čas výpočtu problému.

Prechod	t_min [s]	t_max [s]
in_pause	1000	2000

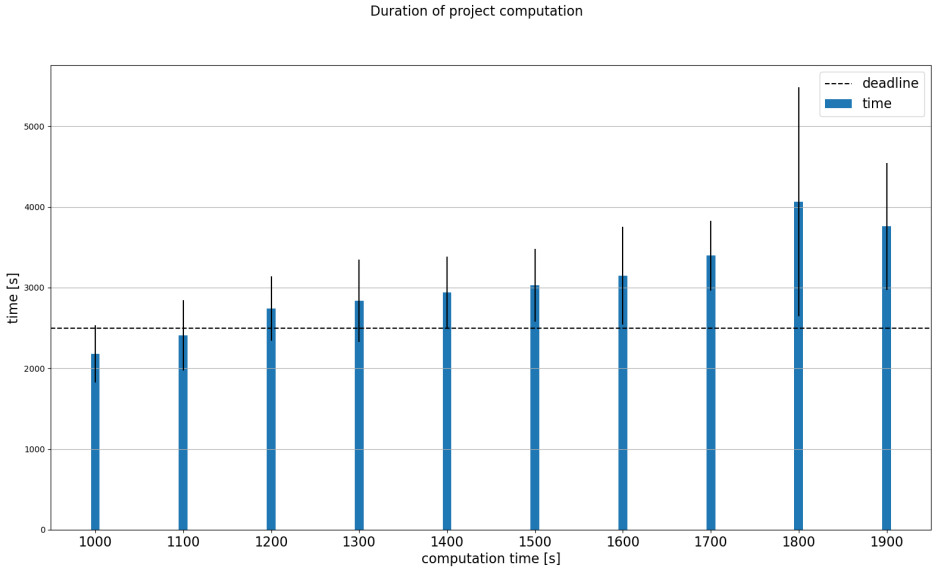
Tab. 12: Nastavenie časovaných prechodov



Obr. 45: Pauza výpočtu používateľom pre rôzne pravdepodobnosti

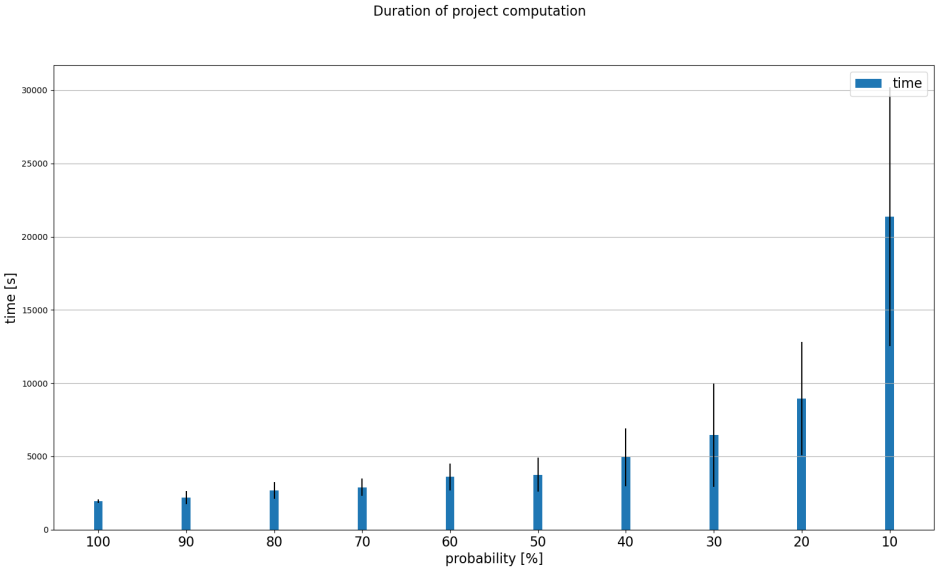
9.5.4 Deadline pre výpočet podúlohy

Vo vytvorenom modeli sú dve možnosti pre vytvorenie takejto simulácie. Prvá je zmenou parametrov časovaných prechodov *compute_1* a *compute_2*, druhá zmenou pravdepodobnosti pri prechodoch *job_running* a *job_crash*. Prvá možnosť je jednoznačná, pretože meníme časy klientov pre výpočet podúlohy. Pri prechodoch stačí zmeniť len hornú hranicu času, teda *t_max*, čím spravíme daného klienta "pomalším". V druhom prípade môžeme pravdepodobnosťou pri prechode *job_crash* znemožniť "druhý" výpočet, čím sa vyžiada nové spustenie výpočtu podúlohy na novom klientovi. Toto znemožnenie teda reprezentuje situáciu, kedy čas výpočtu dostiahol deadline, čiže konečný čas pre dokončenie výpočtu. Obe možnosti si graficky znázorníme. Simulácia bola vykonaná na piatich klientoch a dvoch podúlohách. Prvá možnosť je zobrazená v grafe na obrázku 46, kde je zobrazená závislosť času výpočtu problému pre rôzne zmeny maximálneho času *t_max* pri prechodoch *compute_1* a *compute_2*. Oba parametre *t_max* sú zvyšované o 100 sekúnd, pričom vždy sa simulácia s týmito parametrami opakovala desať krát. Zobrazené hodnoty sú teda stredné hodnoty so smerodajnou odchýlkou. V grafe je zobrazený aj deadline, teda maximálny čas pre výpočet. Všetky stĺpce, teda klienti, ktoré sú pod touto čiarou, prípadne sa ich smerodajná odchýlka pretína s priamkou deadlinu, majú možnosť pre dané časy *t_max* stihnúť výpočet do deadlinu. Sú to stĺpce do hodnoty $t\_max = 1400$, nad touto hodnotou už klienti nemajú možnosť stihnúť výpočet do deadlinu. V grafe obrázku 47 je zobrazená druhá možnosť, čo je teda zmena pravdepodobnosti pri prechode job_running. Táto pravdepodobnosť bola vždy nastavená pre každého klienta rovnako. Na grafe môžeme vidieť exponenciálnu závislosť pravdepodobnosti



Obr. 46: Pauza výpočtu používateľom pre rôzne časy výpočtu

prechodu job_running na čase výpočtu podúlohy. Čím bude táto pravdepodobnosť menšia, tým sa zvýši počet zastavených výpočtov kvôli jeho dlhému výpočtu, a budú sa vytvárať nové požiadavky na jeho výpočet. Pri najmenšej pravdepodobnosti teda predpokladáme, že plánovač neposiela podúlohu len klientom, ktorí majú veľkú pravdepodobnosť pre dokončenie, ale aj takým, ktorí nie sú schopní podúlohu vypočítať do deadlinu. V tomto prípade bude čas pre dokončenie problému vysoký, pretože nebude dostatok klientov ktorí dokončia úlohu v danom čase, a bude sa čakať na ďalších. Toto potvrdzuje opačný efekt malého času pre dokončenie.



Obr. 47: Pauza výpočtu používateľom pre rôzne pravdepodobnosti

10 Záver

Táto práca sa zaoberala modelovaním distribuovaných systémov a ich simuláciou pomocou Petriho sietí. V úvode práce boli zhrnuté základné vlastnosti a charakteristiky distribuovaných systémov a Petriho sietí. Veľký ohľad sa bral na časované Petriho siete, ktoré tvoria samostatnú kapitolu, a to pre ich stochastické vlastnosti. Tieto vlastnosti sú dôležitým prvkom pri modelovaní distribuovaných systémov, a tak boli časované Petriho siete kľúčové pri implementácii modelov v praktickej časti.

Pre implementáciu modelov bola využitá knižnica *PetNetSim*[2]. Táto knižnica bola vyvinutá na ústave Automatizácie a informatiky našej školy. Okrem jednoduchej a výstižnej reprezentácie komponentov Petriho sietí ponúka aj grafický editor, pomocou ktorého boli vytvorené vizuálne modely sietí v praktickej časti. Grafický editor ale nie je vhodný pre tvorenie a simuláciu Petriho sietí väčších rozmerov, ako to bolo pri oboch implementáciách. Preto sú oba modely vytvorené a simulované pomocou skriptu v jazyku *Python* pomocou tejto knižnice. Oba skripty sú dostupné v prílohách práce.

Pri implementácii horizontálne škálovanej webovej aplikácie bola vytvorená a detailne popísaná Petriho sieť tohto modelu. Tento model obsahuje siete load balancera, API brány a mikroslužby. Tieto siete sú vytvorené samostatne, aby bolo zaistené ich klonovanie, ktoré je kľúčové pri oboch implementáciách. Na tomto modeli sa pre rôzne konfigurácie a počty serverov a mikroslužieb vykonali simulácie a overenia pre rôzne situácie. Podarilo sa overiť správanie load balancera a jeho metódy round robin, a tiež prideľovanie požiadaviek jednotlivým mikroslužbám. Ďalej sa podarilo vytvoriť simulácie pre preťaženie servera a mikroslužieb, a tiež simuláciu zlyhania servera, čoho dôsledkom sú kaskádové javy. Výsledky zo simulácií boli spracované buď štatisticky, alebo boli zobrazené na základe časovej závislosti fronty čakajúcich požiadaviek pri mikroslužbách.

Druhý model obsahuje implementáciu veľkého sieťového výpočtového systému, presnejšie platformu *BOINC* s projektom *Folding@home*. Tak ako aj pri prvom modeli, aj pri tomto je detailne popísaná vytvorená Petriho sieť. Tento model sa skladá z troch sietí, a to: rozdelenie problému projektu na časti a ich distribúciu, sieť klienta a porovnanie výsledkov medzi klientami a ich kompozíciu. Tieto siete sú vytvorené samostatne ako pri prvom modeli. Na tomto modeli boli vykonané simulácie pre dve situácie. Podarilo sa vytvoriť simuláciu pre pauzu výpočtu používateľom, a tiež simuláciu deadlinu pre výpočet podúloh. Výsledky z týchto simulácií boli štatisticky spracované a graficky reprezentované.

Správanie implementovaných modelov odpovedá reálnym situáciám, a to hlavne pre stochastické vlastnosti časovaných Petriho sietí a dizajne modelov. Dá sa teda

povedať, že Petriho siete sú dobrým nástrojom pre tvorenie a simuláciu distribuovaných systémov.

11 Zoznam použitej literatúry

[1] Tanenbaum, A. S.; Van Steen, M.: *Distributed systems: principles and paradigms*. distributed-systems.net, třetí vydání, 2017, ISBN 978-90-815406-2-9.

[2] Ladislav, D.: *PetNetSim*. `https://github.com/karna48/petnetsim`, Apr 2021, [Software; accessed 5-May-2021].

[3] PUDER, A.; RÖMER, K.; PILHOFER, F.: *CHAPTER 2 - BASIC CONCEPTS*. In *Distributed Systems Architecture*, editace A. PUDER; K. RÖMER; F. PILHOFER, San Francisco: Morgan Kaufmann, 2006, ISBN 978-1-55860-648-7, s. 7–31, doi:https://doi.org/10.1016/B978-155860648-7/50003-6.

[4] Singh, V. K.: *Key Characteristics of Distributed Systems*. `https://medium.com/system-design-blog/key-characteristics-of-distributed-systems-781c4d92cce3`, 2019, [Online; accessed 10-May-2021].

[5] *Distributed hash table*. `https://en.wikipedia.org/wiki/Distributed_hash_table`, May 2021, [Online; accessed 8-April-2021].

[6] Datta, A.; Gradinariu, M.; Raynal, M.; aj.: *Anonymous publish/subscribe in P2P networks*. In *Proceedings International Parallel and Distributed Processing Symposium*, IEEE Comput. Soc, 2003, ISBN 0-7695-1926-1, s. 8–, doi:10.1109/IPDPS.2003.1213174.

[7] Bause, F.; Kritzinger, P.: *Stochastic Petri Nets - An Introduction to the Theory*. Vieweg, 2002, ISBN 3-528-15535-3.

[8] Šťastný, J.: *Simulace systémů. Elektronická studijní opora*. VUT Brno, 2002.

[9] Šárka Voráčová; Pěnička, M.; Veselý, J.: *Úvod do modelování procesů Petriho sítěmi*. Praha: Česká technika - nakladatelství ČVUT, vyd. 1 vydání, 2008, ISBN 9788001039793.

[10] Bakhrani, H. J.: *Scaling Your Web Application*. `https://medium.com/@harithjaved/scaling-your-web-application-693657ce333c`, Mar 2020, [Online; accessed 25-April-2021].

[11] Villanueva, J. C.: *Comparing Load Balancing Algorithms*. `https://www.jscape.com/blog/load-balancing-algorithms`, Jun 2015, [Online; accessed 20-April-2021].

[12] Fowler, M.; Lewis, J.: *Microservices*. `https://martinfowler.com/articles/microservices.html`, Mar 2014, [Online; accessed 30-April-2021].

[13] Richardson, C.: *Building Microservices: Using an API Gateway*. https://www.nginx.com/blog/building-microservices-using-an-api-gateway/, Jun 2015, [Online; accessed 21-April-2021].

[14] Behara, S.: *Breaking the Monolithic Database in your Microservices Architecture*. https://samirbehara.com/2018/09/03/breaking-the-monolithic-database-in-your-microservices-architecture/, Sep 2018, [Online; accessed 26-April-2021].

[15] Singh, J.: *The What, Why, and How of a Microservices Architecture*. https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9, Jun 2018, [Online; accessed 31-April-2021].

[16] Márton, P.: *Designing a Microservices Architecture for Failure: @RisingStack*. https://blog.risingstack.com/designing-microservices-architecture-for-failure/, Aug 2017, [Online; accessed 30-April-2021].

[17] Anderson, D. P.: *BOINC: A Platform for Volunteer Computing*. *Journal of Grid Computing*, ročník 18, č. 1, Listopad 2019: s. 99–122, doi:10.1007/s10723-019-09497-9.

[18] *How BOINC works*. https://boinc.berkeley.edu/wiki/How_BOINC_works, [Online; accessed 7-May-2021].

[19] *Redundancy and errors*. https://boinc.berkeley.edu/trac/wiki/JobReplication, [Online; accessed 5-May-2021].

[20] Bowman, G.: *2020 in review, and happy new year 2021!* https://foldingathome.org/2021/01/05/2020-in-review-and-happy-new-year-2021/, Jan 2021, [Online; accessed 15-May-2021].

[21] *VUT je mezi českými univerzitami jedničkou v projektu Folding@home*. https://www.vutbr.cz/vut/aktuality-f19528/vut-je-mezi-ceskymi-univerzitami-jednickou-v-projektu-folding-home-d198006, Apr 2020, [Online; accessed 15-May-2021].

[22] Marsan, M. A.; Balbo, G.; Conte, G.; aj.: *Modelling with generalized stochastic petri nets*. Boston: J. Wiley, vyd. 1 vydání, 1995, ISBN 0471930598.

[23] Horvath, G.; Miklos, T.; Sericola, B.: *Analytical and Stochastic Modelling Techniques and Applications: 21st International Conference, ASMTA 2014, Budapest, Hungary, June 30 – July 2, 2014,Proceedings*. číslo 8499 in Programming and Software Engineering, Cham: Springer International Publishing : Imprint: Springer, první vydání, 2014, ISBN 9783319082196.

[24] Zítek, P.: *Simulace dynamických systémů*. Praha: SNTL, první vydání, 1990, ISBN 80-03-00330-x.

[25] van Steen, M.; Tanenbaum, A. S.: *A brief introduction to distributed systems*. *Computing*, ročník 98, č. 10, 2016: s. 967–1009, doi:10.1007/s00607-016-0508-7.

12 Prílohy

Priložený zip archív obsahuje nasledujúce súbory:

- 2021_DP_Duris_Anton_184197_prilohy.zip
  - web_app.py - kód v jazyku Python pre implementáciu horizontálne škálovanej webovej aplikácie
  - boinc.py - kód v jazyku Python pre implementáciu veľkého sieťového výpočtového systému
  - petnetsim - knižnica PetNetSim, aktuálna pre vytvorené modely
  - json files - adresár s json súbormi Petriho sietí
    - load_balancer.json - sieť load balancera a API brány
    - microservice.json - sieť mikroslužby
    - make_tasks.json - sieť rozdelenia problému na časti a ich distribúciu
    - boinc_client.json - sieť klienta
    - compare_tasks.json - sieť porovnania výsledkov medzi klientami a ich kompozícia